Improving scalability of large-scale distributed Spiking Neural Network simulations on High Performance Computing systems using novel architecture-aware streaming hypergraph partitioning.

The University Of Sheffield.

A thesis submitted to the University of Sheffield for the degree of Doctor of Philosophy

Department of Automatic Control and Systems Engineering

Registration number: 160100448

Carlos Fernandez Musoles

September 2020

Abstract

After theory and experimentation, modelling and simulation is regarded as the third pillar of science, helping scientists to further their understanding of a complex system. In recent years there has been a growing scientific focus on computational neuroscience as a means to understand the brain and its functions, with large international projects (*Human Brain Project*, *Brain Activity Map*, *MindScope* and *China Brain Project*) aiming to further our knowledge of high level cognitive functions. They are a testament to the enormous interest, difficulty and importance of solving the mysteries of the brain. Spiking Neural Network (SNN) simulations are widely used in the domain to facilitate experimentation.

Scaling SNN simulations to large networks usually results in more-than-linear increase in computational complexity. The computing resources required at the brain scale simulation far surpass the capabilities of personal computers today. If those demands are to be met, distributed computation models need to be adopted, since there is a slow down of improvements in individual processors speed due to physical limitations on heat dissipation. This is a significant change that requires careful management of the workload in many levels: partition of work, communication and workload balancing, efficient inter-process communication and efficient use of available memory. If large scale neuronal network models are to be run successfully, simulators must consider these, and offer a viable solution to the challenges they pose.

Large scale SNN simulations evidence most of the issues of general HPC systems evident in large distributed computation. Commonly used distribution of workload algorithms (round robin, random and manual allocation) do not take into consideration connectivity locality, which is natural in biological networks, which can lead to increased communication requirements when distributing the simulation in multiple computing nodes. State-of-the-art SNN simulations use dense communication collectives to distribute spike data. The common method of point to point communication in distributed computation is through dense patterns. Sparse communication collectives have been suggested to incur in lower overheads when the application's pattern of communication is sparse. In this work we characterise the bottlenecks on communication-bound SNN simulations and identify communication balance and sparsity as the main contributors to scalability. We propose hypergraph partitioning to distribute neurons along computing nodes to minimise communication (increasing sparsity). A hypergraph is a generalisation of graphs, where a (hyper)edge can link 2 or more vertices at once. Coupled with a novel use of sparse-aware communication collective, computational efficiency increases by up to 40.8 percent points and simulation time reduces by up to 73%, compared to the common round-robin allocation in neuronal simulators.

HPC systems have, by design, highly hierarchical communication network links,

with qualitative differences in communication speed and latency between computing nodes. This can create a mismatch between the distributed simulation communication patterns and the physical capabilities of the hardware. If large distributed simulations are to take full advantage of these systems, the communication properties of the HPC need to be taken into consideration when allocating workload to route frequent, heavy communication through fast network links. Strategies that consider the heterogeneous physical communication capabilities are called *architecture-aware*. After demonstrating that hypergraph partitioning leads to more efficient workload allocation in SNN simulations, this thesis proposes a novel sequential hypergraph partitioning algorithm that incorporates network bandwidth via profiling. This leads to a significant reduction in execution time (up to 14x speedup in synthetic benchmark simulations compared to architecture-agnostic partitioners).

The motivating context of this work is large scale brain simulations, however in the era of social media, large graphs and hypergraphs are increasingly relevant in many other scientific applications. A common feature of such graphs is that they are too big for a single machine to cope, both in terms of performance and memory requirements. State-of-the-art multilevel partitioning has been shown to struggle to scale to large graphs in distributed memory, not just because they take a long time to process, but also because they require full knowledge of the graph (not possible in dynamic graphs) and to fit the graph entirely in memory (not possible for very large graphs). To address those limitations we propose a parallel implementation of our architecture-aware streaming hypergraph partitioning algorithm (HyperPRAW) to model distributed applications. Results demonstrate that HyperPRAW produces consistent speedup over previous streaming approaches that only consider hyperedge overlap (up to 5.2x speedup). Compared to multilevel global partitioner in dense hypergraphs (those with high average cardinality), HyperPRAW is able to produce workload allocations that result in speeding up runtime in a synthetic simulation benchmark (up to 4.3x). HyperPRAW has the potential to scale to very large hypergraphs as it only requires local information to make allocation decisions, with an order of magnitude less memory footprint than global partitioners.

The combined contributions of this thesis lead to a novel, parallel, scalable, streaming hypergraph partitioning algorithm (HyperPRAW) that can be used to help scale large distributed simulations in HPC systems. HyperPRAW helps tackle three of the main scalability challenges: it produces highly balanced distributed computation and communication, minimising idle time between computing nodes; it reduces the communication overhead by placing frequently communicating simulation elements close to each other (where the communication cost is minimal); and it provides a solution with a reasonable memory footprint that allows tackling larger problems than state-of-the-art alternatives such as global multilevel partitioning.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text. This work has not been submitted for any other degree or professional qualification except as specified.

Acknowledgements

I am hugely thankful for the guidance and knowledge provided by Dr. Paul Richmond and the experience and expertise of Prof. Daniel Coca at The University Of Sheffield.

Thank you to Dr. Andrew Turner at the ARCHER HPC supercomputing centre for his continuous support in obtaining computing resources, without which it would not have been possible to carry out most of our experimentations. Big thank you to Mike Croucher and Kenji Takeda for their generous support on the successful Microsoft Azure Sponsorship Offer [1] grant application.

I thank my parents for their support during the early days of my career; even in the distance they have been instrumental.

Finally, I would like to single out my wife Sumitha for keeping me pushing towards the end. Without her encouragement, support and understanding I would have not been able to present this thesis.

This work was supported by BBSRC "The Digital Fruit Fly Brain" under GrantNo.: BB/M025527/1, EPSRC "Accelerating Scientific Discovery with Accelerated Computing" under GrantNo.: EP/N018869/1 and access to ARCHER through an EPSRC Grant (project e582).

[1] https://azure.microsoft.com/en-us/offers/ms-azr-0036p/

Glossary of terms and acronyms

- **HPC**: High Performance Computing.
- **SNN**: Spiking Neuron Network.
- **Complex system**: a system formed by a large number of interactive elements whose combined activity is non-linear.
- **Parallel computing**: synchronous computation across multiple processors, generally within a shared memory context.
- **Distributed computing**: parallel computation across processors connected via communication links (generally without shared memory).
- **NUMA**: Non-Uniform Memory Access.
- **LIF**: Leak integrate-and-fire.
- **MPI**: Message Passing Interface.
- **Process**: software execution construct with its own memory address space.
- **NBX**: Neightbour Exchange communication pattern.
- **PEX**: Personalised Exchange or Personalised Census communication pattern.
- **Hypergraph**: Mathematical generalisation of a graph in which edges can connect two or more vertices.
- **ARCHER**: UK National Supercomputing Service.
- **P2P**: Point-to-Point communication.
- **MVC** model: Macaque Visual Cortex neural model.
- **CM** model: Cortical Microcircuit model.
- **Architecture-aware allocation**: Workload allocation algorithms that take target hardware communication capabilities into account. In contrast to *architecture-agnostic* which do not consider it.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction: challenges to large scale distributed simulations

1.1 Motivation

The term *simulation* is often used as a generic umbrella term that loosely covers various aspects, from mathematical modelling (a simplification or abstraction) to reconstruction and duplication (recreating a system or system's behaviour). In any of its variants, simulations help scientists to further their understanding of a system: they allow them to experiment on a scale that would be difficult or infeasible to do with a real system; act as a test-bed for scientific inquiry, a means to determine relationships by isolating independent variables; or as a proof of concept for further research. After theory and experimentation, modelling and simulation has been labelled as the **third pillar of science**.

In Computational Neuroscience —a multidisciplinary approach that includes computer scientists, mathematicians electrical engineers, physicists and biologists— simulations have become a useful approach to studying the complexity of the brain and its functions. Evidencing this trend, there has been a wide variety of simulators being developed in recent years: general purpose software simulators [65, 91, 42]; hardware solutions [130, 43]; targeted systems [89, 163] and systems to reverse engineer behaviour [203].

Simulations have been key to various neuroscientific discoveries, such as understanding the mechanisms of learning (decoding of temporal information via Short Term Plasticity [41]; learning via Spike Timing-Dependent Plasticity (STDP) [191], understanding the biophysical mechanisms governing STDP [15]), and modelling and understanding neurodegenerative diseases such as epilepsy [146] and its inception (epileptogenesis [120]).

Work that aims to improve Spiking Neural Network (SNN) simulations could benefit engineers in designing better hardware solutions for machine learning. Inspired by biological SNNs, *neuromorphic* hardware [196, 168, 185, 148, 118, 182, 77, 198] seeks to deliver machine learning (pattern recognition, signal processing, image detection) far more energy efficient than deep neural networks [99, 199].

In recent years, there is a growing scientific focus on computational neuroscience as a means to understand the brain and its functions. Just as the 90s was the era of genomics (with the Human Genome Project) and the 2000s for physics (with the CERN Large Hadron Collider), the 2010s-2020s have been marked as the decade

to push neuroscientific discovery [126, 93]. The most visible effort is perhaps the implementation of two big science projects on both sides of the Atlantic: the *Human Brain Project* (HBP) [1] in Europe and the *Brain Activity Map* (BAM) [7] in the United States. Both are set out to further our understanding of the brain, with complementary focuses: whilst the HBP aims to increase knowledge by building brain-scale simulations of the human brain, the BAM strives to create a functional connectome of the brain to understand both structure and function.

Although there has been some criticism to these large projects [207], they are a testament to the enormous interest, difficulty and importance of solving the mysteries of the brain. The sprouting of many other initiatives around the world reinforce that interest: *MindScope* [133] (study of the mouse visual cortex), *China Brain Project* [175] (study of cognitive function with focus on brain disorders), *Brain Mapping by Integrated Neurotechnologies for Disease studies* [2] (Japanese effort to map the brain of a common marmoset monkey), *Israel Brain Technologies* [3] (development of neuroscience-inspired technology) and the creation of political initiatives such as the *BRAIN Initiative* [4] by the USA administration to foster scientific collaboration. These projects are attempts to tackle the problem of understanding the brain that may lie beyond the reach of any individual initiative.

1.2 Need for distributed computing in SNN simulations

Even though full brain simulations may not be the answer to all neuroscientific questions (see [67] for a discussion on epistemological issues with large-scale brain simulations), there is value in modelling and simulating larger networks, and facilitating this level of simulation is a clear goal in the field: the creation of a large-scale brain simulation platform was stated as one of the main aims of the HBP [150].

Scaling simulations to large networks usually results in more-than-linear increase in computational complexity: biological neurons are often connected via synapses to $10^2 - 10^4$ other neurons on average, hence a small increase in the number of neurons leads to several orders of magnitude increase in synapses. Since both neurons and synapses need to be modelled using mathematical equations, doubling the number of neurons on a network typically results in a much higher (greater than 2x) computational demand. In addition, simulating larger systems often requires the use of multiple neuron and synapse models with varying levels of complexity. Moreover, timing constraints (such as the need for simulations to run within reasonable times or meet maximum and minimum delays in synapses to accurately account for synaptic plasticity and learning) increase the computational demands even further.

The computing resources required at the brain scale simulation far surpass the capabilities of personal computers today. Unfortunately, waiting for the compute hardware to catch up, expecting faster CPU clocks, is no longer an option, as evidenced by the slow down of improvements in individual processors speed due to physical limitations on heat dissipation described as the *power wall effect* [12], and

[1] `https://www.humanbrainproject.eu/en_GB`
[2] `http://brainminds.jp/en/`
[3] `http://israelbrain.org/`
[4] `https://www.braininitiative.nih.gov/`

the slow-down of energy efficiency scaling in recent years [99, 151]). Instead, the hardware trend to improve processor speed is to increase the number of computational cores, often of heterogeneous nature, such as a combination of CPUs and GPUs. In the context of distributed discrete event simulations, Fujimoto [80] reports a performance increase from 200 million events-per-second to 500 billion between 2003 and 2013. However, when the measurement is performed relative to the number of cores, the performance gain is less than two-fold (from 138K to 256K). This results exemplify that the computing gain is not being driven by better or faster cores, but by an increase in parallelism and distribution.

A distributed approach to computing requires software programmers and designers to enforce new development paradigms that take parallelism into account. This is a significant change that requires careful management of the workload in many levels: partition of work, communication and workload balancing, efficient inter-process communication and effective use of available memory (within shared and non shared memory spaces). If large scale neuronal network models are to be run successfully, simulators must consider these, and offer a viable solution to the challenges they pose.

1.3 Challenges to distributed simulations

The main identified challenges for scalable performance of general parallel and distributed simulations are **workload balancing** and **communication overhead** [59, 80]. The current trend of heterogeneity of computing hardware (CPU, GPU and other custom processor types) adds to those challenges with variable communication costs between nodes, different computational speeds and capabilities or access to uneven amount of memory per node. Furthermore, this heterogeneity poses new difficulties, such as Non-Uniform-Memory-Access (NUMA) which requires structured access patterns to maximise cache utilisation, efficient low-overhead data structures to reduce memory footprint, and good mapping between tasks and workload.

The need for distributed computation brings three main challenges to SNN simulations:

- Complex **workload balance**: motivated by the increasingly heterogeneous nature of HPC computers nowadays, together with the heterogeneous nature of biological neuronal networks with elements requiring uneven computing time.
- Increased **communication overhead**: elements in biological networks (neurons) are highly interconnected; when simulations are run in parallel, the probability of connected neurons to reside within the same node is reduced and hence communication is increased when propagating information between neurons.
- Large **memory requirements**: Brain-scale biological simulations will require processing and running networks that are too big to fit in memory. Distribution and access of data structures, as well as efficiency of algorithms that deal with the distribution, is required.

If brain-size simulations are to be run in efficiently, distributed and parallel simulations must address those challenges effectively.

1.3.1 Workload balancing

To solve problems faster, one can break down the problem into smaller parts that are worked on independently and simultaneously by different processors. Load balancing refers to the assignment of parts of the work to computational resources that are underutilised [20] in order to reduce overall execution time.

To maximise the utilisation of the computing power of a hardware architecture, the time any of the processors spends idle should be minimised —as it is time a processor could have spent in useful computation. A good balanced application needs to solve these two elements: division of workload and assignment of work to available processes. Except so-called *embarrassingly parallel problems*, most problems do not have a straightforward answer to at least one of the elements. This is specially true in the case of dynamic applications, where the workload changes over time.

Large simulations of scale-free networks (with power-law node connectivity) with irregular topologies and skewed node degree distributions (such as certain SNN) have been shown to be difficult to parallelise, due to load imbalances [80] (consequence of highly skewed communication overheads due to skewed connectivity distribution) and severe communication overheads (difficult to partition, leading to required communication between parallel computing nodes).

1.3.2 Communication overhead

In distributed applications, higher parallelism must be traded off against communication overhead [224]: local updates in a process may need to be communicated to other processes, or aggregated to determine the final answer. As the number of processes used to compute a global problem grows, the potential communication between those processes grows super-linearly.

In the context of parallel SNN simulations, Zenke and Gerstner [229] identify a hard boundary to the speed up by parallelism due to inter-process communications and spike propagation as the number of processes is increased. Parallelisation saturates at some point [8], when increasing number of processes does not yield lower simulation runtime, as the communication costs become more relevant than computation costs.

To further improve performance via increased parallelisation, the impact of communication overhead must be minimised, either hiding communication costs or by reducing the overall communication required. One approach is to carefully divide the problem domain so less partitions need to communicate —chapter 3 explores this idea.

Particularly important is communication in neuronal simulations using electric synapses [96], where the state of pre and post-synaptic neurons cannot be solved independently and hence each neuron pair needs to communicate on each simulation step. This is also the case for neuromodulated signals [178], where the synapse needs to be informed about a population-based signal that, in distributed simulations, resides in a different process.

1.3.3 Memory requirements

Large scale biological simulations will require processing and running networks that are too big to fit in a single memory space. Work on efficient memory distri-

bution for parallel SNN simulations has been carried out [136, 137, 124, 132], with the focus on optimising simulation data structures that already reside in remote processes.

Tackling the workload and communication challenges requires processing large networks to efficiently distribute workload in a way that minimises communication overhead. This is a non trivial process that requires applying optimisation algorithms such as graph partitioning (explored throughout this thesis). However, at very large scales, those algorithms have to deal with memory requirements that surpass the capacity of the system. Memory efficient algorithms are required to enable scalability.

1.4 Challenges to heterogeneous HPC

There are two types of heterogeneity commonly found in HPC systems: communication and computing heterogeneity. The former is due to the varied physical connectivity between computing nodes. The latter is a result of having different computing cores (CPU, GPU, Vector Processing Units, other custom accelerators) present across nodes, or even within the same node.

HPC systems achieve high performance through parallelism and distribution. By the distributed nature of their architectures, there is a level of communication heterogeneity between any two running processes in the system (fast shared-memory communication between processes within the same computing node versus slower electric and optic connections between remote nodes). Take as an example the architecture of ARCHER[5], the UK National Supercomputer Service. Each computing node has two 12-core Intel Ivy Bridge processor. Four nodes are connected to an Aries router, 188 nodes are grouped into a cabinet; and two cabinets make up a group. There are all-to-all electric connections between nodes in the same group and all-to-all optical connections between different groups. This connectivity pattern (comparable to other HPC systems) leads to different connectivity bandwidths and latencies between processes, depending on where they are hosted.

This complex connectivity pattern is often a bottleneck for speed of communications [201]. Hence careful mapping of virtual topology (what needs to be computed) to the physical topology (the hardware architecture) is required for an optimal execution to maximise the speed of communication between processes.

McCalpin [155] discusses the different rates of improvement in computing per component, comparing network bandwidth and latency and memory bandwidth and latency to the gain in computation speed (measured in floating point operations per second, FLOPS). He shows how, comparatively, gains in computational power (FLOPS) are at a much higher rate than network and memory latency (x30 and x100 respectively), and less acutely gains on bandwidth. Since computing speed is the fastest growing factor, it follows that to achieve faster parallel computations software must maximise the time it spends computing whilst reducing the time spent communicating.

Each processing unit type is more suited for certain types of computation. CPUs are designed for low latency tasks, whereas GPUs are ideal for high throughput, high latency ones. This should inform the division of workload to optimise the computing

[5] `http://archer.ac.uk/`

throughput of the overall architecture. The hardware trend of increasing parallelism to achieve better performance [12] highlights the importance of this consideration.

1.4.1 Other challenges to HPC

In addition to the challenges brought by heterogeneity, there are other difficult problems in HPC computation that remain an active area of research, such as memory access patterns [20], parallel-efficient data structures [136] and energy efficiency [99, 151]. Although these are important challenges, they lie outside the scope of this thesis.

1.5 Contributions of the thesis

The overall aim of this thesis is to address the performance shortcomings of large distributed simulations, in particular the poor scalability of the communication requirements due to the allocation of workload across distributed computing nodes. The thesis evaluates using hypergraphs to model distributed applications and the impact of incorporating hardware capabilities as part of the workload allocation mechanism.

Formally, the **research question** the thesis attempts to answer is: *can hypergraph models and architecture-aware partitioning algorithms improve scalability of large distributed SNN simulations on High Performance (HPC) systems?*

Large distributed simulations are often communication bound whose scalability, required due to the size of the problem, is limited by the increase in communication needed to synchronise data across computing nodes. The heterogeneous nature of HPC systems makes this problem more difficult with varied communication bandwidth and computational capabilities resulting in uneven costs of computation and communication across nodes.

This work evidences and proposes solutions for the three main computational challenges of HPC scaling, namely: keeping workload balance for efficient use of allocated computing nodes; reducing the communication overhead of large-scale distributed applications; and creating optimal strategies to adapt to communication heterogeneity in HPC systems to improve distributed application performance. All whilst considering the memory requirements associated with optimisation algorithms at large scale.

The contributions of this work are summarised here:

- **C1**: (Chapter 3) Demonstrate that communication sparsity between computing nodes drives performance in communication-bound SNN simulations.
- **C2**: (Chapter 3) Produce sparser communication patterns (up to 90% less Average Runtime Neighbours and up to 80% less volume of data) by modelling SNN simulations as a hypergraph and using partitioning algorithms.
- **C3**: (Chapter 3) Reduce the overheads on the three phases of P2P communication (synchronisation, handshake and data exchange) in SNN simulations with the use of dynamic sparse communication patterns, resulting in more balanced inter-process communication (up to 90% less implicit synchronisation

time), faster simulation runtime (up to 73% less time) and more computational efficiency (up to 40.8 percentage points more time spent in computing).

- **C4**: (Chapter 3) Develop a novel framework for testing communication and workload allocation strategies Spiking Neural Network simulations.
- **C5**: (Chapter 4) Speedup runtime (up to 14x) in distributed applications modelled as a hypergraph by mapping application communication patterns to hardware architecture communication heterogeneity.
- **C6**: (Chapter 4) Propose a novel cost function for restreaming hypergraph partitioning to balance heterogeneous communication costs and workload balance.
- **C7**: (Chapter 4) Develop a synthetic benchmark for modelling distributed applications in heterogeneous HPC systems to aid the evaluation of workload allocation strategies.
- **C8**: (Chapter 4) Demonstrate that using a novel global communication metric that guides the refinement of restreaming partitioning improves the mapping of application communication patterns and hardware architecture heterogeneity.
- **C9**: (Chapter 5) Implement a novel parametric hypergraph generator to facilitate benchmarking of complex distributed applications modelled as hypergraphs.
- **C10**: (Chapter 6) Develop a novel architecture-aware streaming hypergraph partitioning algorithm that can run in distributed streams efficiently without significant loss in quality.
- **C11**: (Chapter 6) Demonstrate the impact that process workload balance and multistreaming start has on the efficacy of hypergraph partitioning-based work allocation.
- **C12**: (Chapter 6) Reduce runtime on modelled distributed applications using parallel streaming hypergraph partitioning (up to 5.2x over other streaming partitioners, and up to 4.3x on dense hypergraphs over global partitioners).
- **C13**: (Chapter 6) Efficient and scalable streaming hypergraph partitioning with reduced memory footprint compared to global state-of-the-art hypergraph partitioning algorithm.

As a result of the work in this thesis, I developed and published [6] a novel multistreaming hypergraph partitioning algorithm (*HyperPRAW*) that maps distributed applications workload to computing nodes in HPC systems that tackles the main identified challenges:

- Keeps workload balance by maintaining the ratio between maximum and average workload amongst computing nodes to a user-defined threshold. By incorporating a workload parameter within the allocation function, it reduces the workload imbalances between any two computing nodes, which reduces both computation and communication idle time.

[6] `https://github.com/cfmusoles/hyperPraw`

- Minimises distributed communication by taking advantage of computation locality, i.e. placing communicating elements of the application in the same computing node to avoid inter-node messaging. This is achieved by modelling the application as a hypergraph that is then partitioned and mapped to nodes to minimise relevant metrics.

- Optimises the distributed communication of the application by using hardware bandwidth information (from profiling) to modify the workload allocation in order to place highly connected computing elements close to each other.

- HyperPRAW, as a parallel streaming partitioner, has an order of magnitude less memory requirements than global partitioner counterparts, which allows for further scalability.

1.6 Thesis organisation

The remainder of the thesis is organised as follows.

- Chapter 2 surveys the literature in the core areas in which this work is based. It offers an introduction to computational neuroscience and Spiking Neural Network simulations. Previous attempts at distributed computing and work on optimisation of communication in such simulations are discussed and compared. The industry-standard distributed communication MPI library and common communication patterns are described. Finally, in the context of workload allocation, the graph and hypergraph partitioning problems are presented as a multi-objective solution to optimise load balancing and communication in parallel and distributed applications. The hypergraph partitioning state-of-the-art is reviewed to identify gaps and opportunities in relation to HPC challenges.

- Chapter 3 evidences the bottleneck to scaling communication-bound SNN simulations and presents a novel approach to minimise communication overheads by modelling SNN as hypergraphs which can be partitioned with off-the-shelf partitioning algorithms. Increasing the sparsity of inter-process communication enables sparse collectives to more efficiently handle data synchronisation requirements for distributed SNN simulations, resulting in reduced communication overhead and improved computational efficiency.

- High Performance Computing (HPC) systems have very heterogeneous communication capabilities consequence of their hierarchical architecture, resulting in wide variations in the cost of communications between compute units. Large distributed SNN simulations and other large scale computations have communication patterns that may not match the capabilities of the HPC system. Chapter 4 shows that this mismatch results in performance degradation, and presents improvements that are brought to distributed applications when incorporating hardware bandwidth of the targeted computer system (the one on which the application will run on) during the workload allocation process. Architecture-awareness allows the communication to be weighed by the relative bandwidth leveraging faster connections and minimising the use of slow

ones, resulting in an overall reduction of communication time. This work proposes a novel sequential architecture-aware hypergraph streaming algorithm to aid workload distribution for large scale distributed simulations.

- To better understand the impact of the properties of a hypergraph in graph tasks such as partitioning, it is important to sample the space of possible hypergraphs in a way that different categories of hypergraphs are represented. Chapter 5 proposes a novel hypergraph generator algorithm that can be parameterised to produce custom hypergraphs with defined vertex and hyperedge degree distributions, underlying number of clusters, size and cardinality. This hypergraph generator furthers benchmarking strategies that work over hypergraphs, such as hypergraph partitioning used to allocate workload in distributed simulations.

- The earlier work in chapter 4 shows that partitioning the hypergraph that models a distributed application communication patterns employing network bandwidth data leads to significant runtime speedup (up to 14x) compared to architecture-agnostic partitioner. However, as a sequential, multi-pass algorithm, it operates under strong performance constraints, which result in long partitioning times (hours for large hypergraphs). Chapter 6 explores ways of mitigating the above limitations and proposes a parallel streaming implementation for architecture-aware hypergraph partitioning (HyperPRAW). *HyperPRAW* is a parallel partitioner, with improved scalability with respect to global partitioners, which are known to scale poorly for large graphs.

- Chapter 7 sumarises the contributions made in the thesis.

1.7 Published work to date

This thesis expands on the work I have already published. [72] established the potential gains that are brought by modelling Spiking Neural Network (SNN) simulations as graphs and applying partitioning algorithms to guide the workload allocation. The results show a reduction in the communication volume and a consequent decreased simulation runtime. The Frontiers in Neuroinformatics paper [71] (first presented in the NEST Conference 2019 [73]), explores further models of SNN simulations, along with the use sparse data exchange algorithms instead of traditional point to point communication patterns, demonstrating a significant improvement in computational efficiency —the time a simulation spends in useful computation. More broadly, [74][7] demonstrates the impact of HPC heterogeneity on large distributed applications runtime and how a partitioning algorithm that is architecture-aware can lead to better performing workload allocations.

This previous work sets the basis for HyperPRAW, the proposed novel parallel architecture-aware hypergraph partitioning algorithm, targeted at optimising workload allocation for large-scale distributed applications such as SNN simulations or large sparse tensor multiplications, a common scientific kernel.

[7]DOI: `https://doi.org/10.1145/3337821.3337876`

Chapter 2

Literature Review

This chapter sets the theoretical basis for the rest of the thesis. It describes the fundamentals of computational neuroscience and SNN simulations as a case study for the general problem of reducing communication overhead in distributed scientific agent-based simulations that was introduced in chapter 1. Previous approaches to parallel and distributed SNN simulators are discussed, particularly in how they deal with the HPC challenges (communication overhead, workload balance and heterogeneity). In depth review of current optimisation of spike propagation and workload allocation is offered.

After introducing SNN simulations, I outline concepts around parallelism and distributed computation such as task and data parallelism with respect to distributed simulations. As a de-facto standard distributed communication library, MPI is briefly described, highlighting relevant concepts such as collective versus point to point communications, synchronous and asynchronous modes, barriers, neighbouring primitives and one-sided communication. Within the context of MPI, topology mapping is discussed as an attempt to map application communication patterns to hardware architectures.

Hypergraphs have been shown to represent well communication in parallel applications [63, 61]. Graph and hypergraph partitioning models are presented as theoretical frameworks to be used in optimising load balancing and communication optimisation in distributed applications in general and in SNN simulations in particular. The challenges and nuances of hypergraph partitioning with respect to parallel application modelling are reviewed: edge-centric versus vertex-centric partitioning, streaming and global approaches, scaling of parallel partitioning.

2.1 Computational Neuroscience

2.1.1 Modelling neural networks

Computational neuroscience aids in the understanding of the behaviour of biological neuron networks by building and executing mathematical models that describe them. From this perspective, a neuron is seen as a cell that contains three functional parts: dendrite, input connections to the neuron; soma, the body; and axon, the output connections to other neurons. Neurons are connected using synapses and communicate information via spikes, which are sudden changes in membrane potential (voltage) due to opening and closing of ion channels, resulting in polarisation or

depolarisation of the cell. When a neuron receives input above a threshold it causes the neuron to *spike* (or *fire*) and carry on the message to its output connections. In this type of networks, the message is temporally-encoded (the frequency and interval between spikes), rather than in the actual content of the spike transmission (all spikes are all equal). This networks are referred to as Spiking Neural Networks (SNN).

Early work by Hodgkin [107] set the foundation for biophysical models of neurons. In their model, neuron dynamics are described by ionic equations, which determine membrane potential of the cell based on the conductance of multiple ion channels (sodium, potassium and *leaky*, which at the time was used to explain channels not explicitly modelled).

Often the biophysical model based on conductance is too complex and expensive for execution in real-time. Phenomenological models based on modelling spikes, rather than the biophysical components of the cell, provide a more economical yet still useful approach. The model most widely used is the Leaky Integrate-and-Fire (LIF) [85]. LIF equations include a capacitor u, a resistor R and a current I.

$$\tau_m \frac{du}{dt} = \text{-}u(t) + R\ I(t) \tag{2.1}$$

Functionally, LIF neurons require two elements: an equation to describe the evolution of the membrane potential (equation 2.1) and a process to generate spikes. When the membrane voltage reaches certain threshold, a spike is generated and propagated to all connected output neurons. The membrane voltage is then reset to a resting value. Some models include refractory period in which the neuron is unresponsive after resetting. In absence of input, the membrane potential decays to the resting value using linear or non-linear functions.

2.1.2 SNN simulation

The use of simulators allows researchers to detach themselves from the low-level implementation details [60] such as memory management, data structures and communication patterns, and instead focus on specific areas (equations governing neurons and synapses, larger connectivity networks, learning) which in turn facilitates experimentation.

There are two large groups of simulators: software simulators that run in general computers and hardware simulators built on specific architectures. This overview limits its scope to open, currently supported, simulators. For a more in-depth review of SNN simulators, see [37, 36, 60, 28].

NEURON [42] is one of the oldest neuronal simulators, widely used by the neuroscientific community, with over 2300 citations[1]. Originally designed as a tool for the simulation of biologically realistic neurons, it has grown to include artificial neuron models such as LIF and multiple integration methods to choose from.

NEST [86] is a well supported simulator that focuses on the dynamics of a simulation of spiking neural networks without morphological details. It is part of the NEST Initiative[2], a non-profit organisation that aims to facilitate collaborative efforts to support the development of large-scale simulations of biological neural

[1] `http://www.neuron.yale.edu/neuron/static/bib/usednrn.html` accessed on 1/Apr/2020.
[2] `http://www.nest-initiative.org/`

networks. NEST offers a great variety of neuron and synapse models, connectivity patterns and runtime access to neuron states.

Nengo [203] is a simulator that helps in the creation of population-coding neurons for the Neural Engineering Framework (NEF) [202]. Rather than attempting to model and replicate the behaviour of a particular neuronal network, the goal of NEF is to reverse engineer a particular function; given the function to be performed, and a neuronal population as substrate, NEF aims to solve the connection weights to allow the network to produce the desired behaviour.

In the last few years, other approaches have been proposed to fit different user needs: high performing GPU simulators [22, 228, 25, 208, 204, 163, 164, 159], ease to use simulator builders [92] and those with narrower scopes such as *Auryn* [229] (focused on mid-sized networks, particularly to study synaptic plasticity), *ANNarchy* [217] (tailored for cognitive modelling, where mean-firing rate neurons may be appropriate) and *Neurokernel* [89] (rather than a standard simulator, it is a software platform that facilitates the design and simulation of fruit fly brain regions, taking advantage of GPU acceleration).

On the hardware side, there are several avenues being actively researched to simulate SNN, with advantages and disadvantages to each approach: analogue circuits [199, 118] (much faster and realistic but less programmable), custom massively parallel hardware [130] (highly parallel but lack of flexibility) and FPGAs [79, 43, 185, 50] (flexible but with slow development cycle).

2.2 Parallelism and distribution

The need for parallel and distributed computation to scale up simulations has been established by the power wall effect [12]. This parallelism, however, can be implemented in multiple ways. Taking advantage of new parallel and distributed architectures poses new challenges to computation in general, and simulations specifically. The body of knowledge that can be used to tackle those challenges includes techniques from graph analysis, scheduling, load balancing and topology mapping. Current simulators offer solutions to some of those problems but not all, demonstrating a gap in research for scalable SNN simulations of brain-size.

2.2.1 Types of parallelism and computational models

Parallelism can be seen from multiple perspectives (on software, on hardware, by memory access type) and hence it is difficult to portray a single taxonomy. Modern computers further blur the line by having a combination of heterogeneous processors. However, Flynn's taxonomy of parallelism [20] still offers a good overview of the different computing paradigms available, based on the number of instructions executed per cycle and the data targeted by them: Single Instruction Single Data (one core processors), Single Instruction Multiple Data (vector processors, SMs in a GPU), Multiple Instruction Single Data (for fault tolerance systems) and Multiple Instructions Multiple Data (GPUs). MIMD can be refined into shared-memory MIMD (Uniform Memory Access and Non-Uniform Memory Access, NUMA) and distributed memory (can only communicate via messages). SIMD is often referred to as Single Instruction Multiple Threads (SIMT) when a single function is executed over multiple data but with multiple threads, as is the case in current generation

GPUs. Table 2.1 summarises the main categories, offering general characteristics and performance critical aspects of various parallelisation strategies.

Table 2.1: Approaches to parallelism

	MIMD 	**SIMD** 	**SIMT** 
Paradigm	Task-parallel Thread-parallel	Data-parallel	Data-parallel
Characteristics	Multiple processes executing multiple functions	Single execution over large volumes of data	Single function – multiple threads over large volumes of data Threads in lockstep
Performance critical	Communication overhead Synchronisation	Memory bandwidth Data volume	Memory access Thread divergence Synchronisation
Hardware example	Multicore CPU Computer cluster	Intel Xeon Phi	NVIDA GPUs

From a programmatic perspective, there are three distinct levels of parallelism: instruction level (CPUs executing multiple instructions per clock cycle), task-level, data-level. Instruction Level Parallelism is addressed by compilers to make use of pipelining and therefore it is often out of the scope of parallel application programmers. The most relevant for parallel applications are task and data-level parallelisms.

Task-level parallelism refers to multiple processes running concurrently to execute more than one function or task. From a practical view, these processes are treated as different applications, and they must use formal messaging to communicate (normally MPI[3]). It is limited by the overhead in communication and synchronisation between processes. Multicore CPUs are well suited for this type of parallelism (taking advantage of the low latency of each individual CPU), as are clusters of distributed computers where each computer is treated as a node that can process independently from others, and communicate via network connections. Applications tuned for task-level parallelism are latency oriented, i.e. finishing individual tasks as quickly as possible.

Data-level parallelism refers to the execution of a single instruction or function (SIMD and SIMT) over a large set of data. Usually limited by memory bandwidth and access patterns to minimise the data transfer costs. Vector processors and GPUs operate under this paradigm to parallelise execution. Applications tuned for data-level parallelism are throughput oriented rather than latency, i.e. perform as many tasks *at any given time* as possible.

According to the physical access to memory in distributed applications, there are three computational models: *shared memory* (where processes have direct access to each other's memory spaces), *message passing* (where processes have to shared memory via messaging), and *hybrid models*. Although the limits between shared and remote memory are being blurred (custom hardware support that allows remote memory access), the most applicable model for large-scale distributed systems is hybrid; processes within a computing node share memory but intra-node communication is done via message passing. Therefore message passing and communication overhead are key for scalable distributed computation.

[3] *Message Passing Interface*, a standard protocol to share messages

Distribution is the process of dividing global computation into independent, executable fragments that can be run in non-shared memory devices, such as a network of computers or a multiprocessor CPU. In distributed systems, each device runs its own instance of the program, but together they may collaborate to solve a bigger problem.

Computer engineers have made efforts to design circuits that match data-level parallelism at the hardware level. Two prominent examples of this type of hardware are: vector processor unit (VPU) and Graphic Processing Unit (GPU). The use of hardware accelerators in HPC systems increases computation heterogeneity and justifies the application of adequate scheduling and load balancing algorithms to deal with workload allocation of parallel and distributed applications.

Most HPC systems are design as clusters of nodes with distributed memory. This configuration falls under the category of MIMD model of execution with communication happening via messaging -though the advent of Remote Direct Memory Access (RDMA)[4] technology blurs the line between remote and local memory access. The clustered nature of its systems makes task parallelism more relevant to HPCs, with potential for local data-level parallelism. Therefore, when looking at communication performance of large-distributed HPC systems, the focus is on communication via local and remote messaging, for which the Message Passing Interface (MPI) is the de facto standard.

2.3 Distributed communication: Message Passing Interface

The Message Passing Interface (MPI)[5] is a community defined standard for message passing that attempts to gather best features of other message passing systems. It does not contain implementation details but rather it is an interface that implementers need to follow if they want to meet the standards. It specifically defines methods for message passing computational model, and it is the *de facto* standard in distributed communication. One feature that makes it ideal for distributed systems is its support for heterogeneous networks (joining together workstations from different hardware like in HPC systems).

2.3.1 MPI concepts

In MPI, a **process** has its own parcel of memory addresses and communication entails copying the data from one process to the portion of memory addresses controlled by the receiving process. A process may contain one or more threads, but it always has its own address space.

Sending and receiving messages involves the use of buffers for both sender and receiver (sender places data in a buffer, when receiver gets the message it prepares a receiver buffer with the information contained in the message: length, data structure type, etc.). Therefore, when two processes communicate, there is a necessary handshake between them, which together with the need to create buffers and copy data to and from them result in a communication overhead per communication.

[4] `http://www.rdmaconsortium.org/`

[5] Maintained by the MPI Forum `https://www.mpi-forum.org/`

Initially all processes belong to the same **communicator** group, and within a group processes are identified with a **rank**. Custom communicator groups can be defined to target multiple processes with one send operation.

MPI supports **Blocking** and **non-blocking** send modes, depending on whether it stops program execution or not, useful to probe for multiple messages without having to wait for a specific one; as well as multiple **communication modes**: standard (sender blocks until buffer can be reused), synchronous (send blocks until message is received), and more user-controllable ready and buffered modes. These give the parallel application programmer flexibility to define the best communication pattern to suit their computation model.

Aside from allowing to send messages from a sender to a specific receiver or communicator group, MPI defines common collective operations within a communicator: gather, scatter, reduce, broadcast, all to all (gather and scatter). This are often heavily optimised in any MPI implementation and should be preferred to custom implementations as they often incur in less communication overhead. Figure 2.1 summarises the common collectives. More advanced collectives have been proposed over the years, some of which specifically designed to deal with neighbouring and sparse communication -see section 2.3.2 for a review.

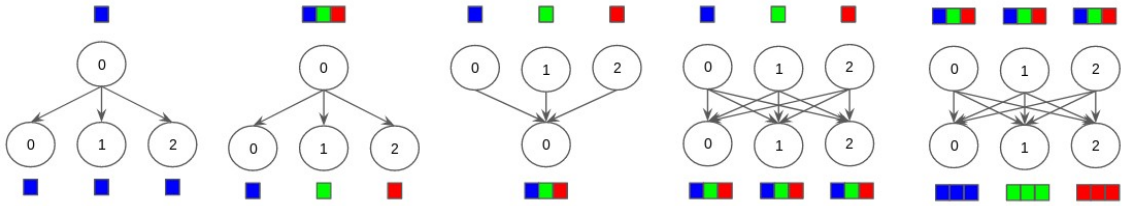


Figure 2.1: MPI Collective communications. From left to right: *Broadcast*, *Scatter*, *Gather*, *All gather* and *All to all*. The colours represent unique pieces of data to be sent. The arrows represent the directionality of messages. Numbered circles indicate distinct MPI processes.

Multi-process applications using MPI have a *virtual topology* that represents the relationship between executing processes. In order to run the application, those processes need to be mapped to the *physical topology* of the computer running the program.

2.3.2 Large and sparse distributed communication

Thakur et al. [206] argue that at the peta and exascale computing, current communication patterns in MPI (the standard library for distributed computing) struggle to scale. The main two challenges are memory footprint of irregular collective operations and poor scaling of collective communications [17]. The former is due to the requirement to define buffers the size of the number of processes as function arguments in every irregular collective call. The latter is a consequence of the nature of all to all communications, each process sending data to all others, with little opportunity for optimisation —although work has been done to improve performance on specific hardware [149, 4]. Since brain scale simulations require large scale distributed architectures, SNN simulations have to deal with those issues too.

Communication in SNN and many other complex system simulations falls into the category of *census*, a common parallel programming function in which a process receives a piece of data from each of all other processes. Each process knows to whom it needs to send data (any process hosting post-synaptic targets of spiking local neurons), but has no information as from whom it is going to receive data (non-local firing neurons). Personalised census or personalised exchange (*PEX*) is the most basic implementation of census [111] in which communication occurs in two steps: 1) interprocess handshake and 2) send and receive data. During handshake, processes inform their targets that they will be sending data to them. In the second phase, each process post data and listens to messages only from those processes.

Often collective communications become the bottleneck on large-scale distributed applications as their complexity grow supra-linearly with the number of processes [172, 58, 143]. There are multiple approaches to dealing with the bottleneck: RDMA technology, non-blocking collectives, topology collectives and novel sparse specific communication algorithms.

Remote Direct Memory Access (RDMA) is introduced to help bridging the gap between shared memory systems and remote memory systems in which thanks to hardware compatible systems, processes can access predefined areas of a remote process memory space. This helps overlap computation and communication [95, 4].

Non-blocking collectives combine the benefits from point-to-point non-blocking communications (overlap of communication and computation) with the elegant and high performing benefits of collective communications. By not blocking they allow less idle time in distributed computations. Effective use of non-blocking collectives as an optimised communication collective is demonstrated by Kumar et al. [135] on the Blue Gene/P Supercomputer.

Hoefler et al. [113, 109] presents new neighbour collectives using virtual topology for sparse collective communications for gather, all to all and reduce. Collectives reflect more accurately the communication patterns of scientific distributed applications (highly localised), resulting in sparse communication. This type of comm pattern (collectives with only a subset of processes) was not supported before.

Neighbourhood Exchange (*NBX*) belongs to the class of *Dynamic Sparse Data Exchange* (DSDE) algorithms proposed by Hoefler et al. [111]. They are based on non-blocking collectives to overlap communication handshake and data transfer. For sparse communication, Hoefler et al. [111] demonstrate that *NBX* performs better than *PEX*, and DSDE is already in used in libraries [116].

NBX is depicted and compared to *PEX* in Figure 2.2. First, each process sends data asynchronously (non-blocking) to all its targets. Whilst the messages are being delivered, processes probe for incoming messages. Once a sender process is notified that its messages have been received, it places an asynchronous barrier and continues to probe for incoming messages until all other processes have reached the barrier (with an MPI_Ibarrier call), signalling the end of the communication phase. The use of an asynchronous barrier removes the need for an explicit handshake, effectively allowing the data exchange to start without explicit knowledge from the receiver.

2.3.3 Topology mapping

When dividing workload to take advantage of parallel computing, there is a trade off between load balancing and communication overhead [97, 229]. As work

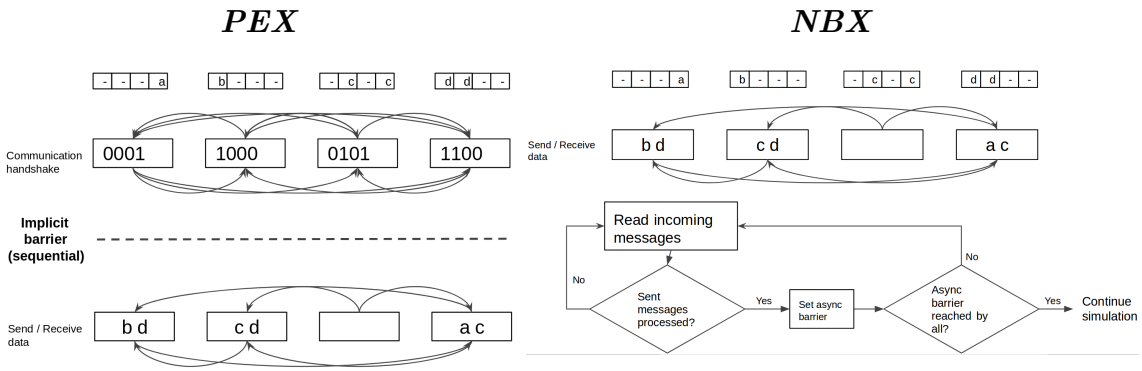


Figure 2.2: Point-to-point communication strategies compared. Left: Personalised Exchange (*PEX*) involves an explicit handshake to coordinate communication intention amongst processes before exchanging data. First, processes share metadata in an all to all manner; in a second round, processes send data only to those that need to receive spiking data. Right: Neighbour Exchange (*NBX*) overlaps both phases with the use of asynchronous barrier, which allows processes to send and receive spiking data while waiting for notification that other processes have completed sending data, minimising synchronisation time.

is divided amongst computational nodes, each processor has to do less work, but in an interconnected simulation this would incur into more inter-node communication. Hence, simulation workload placement onto physical nodes becomes an optimisation exercise where elements with higher levels of communication are placed in cores with high speed connectivity.

Most MPI implementations offer various levels of virtual to physical topology mapping. The out-of-the-box solutions are often insufficient, however this is an active area of research. This section surveys some proposed improvements.

Rodrigues et al. [184] presents a communication-aware computation to process mapping that uses Dual Recursive Bipartitioning to reduce the amount of communication in parallel applications. They map process to core to minimise communication cost by taking into consideration relative speed of intra-chip, inter chip and inter node communication.

The mapping from virtual to physical topology is a non trivial problem that can be modelled as the *graph embedding problem* [181]. This problem is a generalisation of the graph partitioning problem where the objective function to minimise is the sum of the physical communication cost (given by a communication cost graph) of all edges in the process graph. Rather than offering a solution, MPI exposes an interface for developers to incorporate their own [108]. This way MPI users can tailor solutions that make the application independent from the runtime system. Examples of simple, affordable heuristics are [100, 209, 23, 181], based on Kerningham-Lin (KL) improvement (see section 2.6.2) and considering processor hierarchy. To inform the physical topology graph it is common to use profiling tools [157, 23, 181].

Hoefler and Snir [112] developed LibTopoMap[6], a library that supports process topology mapping to any arbitrary heterogeneous network topology. It offers multiple mapping functions, from a greedy heuristic to recursive bisection. The target is to minimise dilation (average length of path taken to go from two different processes)

[6] `http://htor.inf.ethz.ch/research/mpitopo/libtopomap/`

and maximum congestion (traffic link over its capacity).

In addition to communication costs, [157] takes into account memory cache utilisation, an important feature for NUMA architectures. The assignment of processes to cores is based on the amount of data they exchange. An improvement is presented by [121], where hardware hierarchically is represented in a balanced tree-like structure. It produces good mappings for applications with hierarchical communication patterns.

MPI topology mapping has been used for load balancing in chemical simulations [24]. Communication performance is improved by using architecture shape (3d mesh or torus) to map communicating tasks on processors that are physically close together. The algorithm assigns invariant static workload to processors and leaves the code responsible for calculating force between static pairs to be dynamically balanced based on preference tables.

Bokhari [30] suggests the mapping problem can be modelled as the *Quadratic Assignment Problem* (QAP). In order to optimise MPI communication in unstructured computational fluid dynamics, [34] uses QAP heuristics to map processes to cores. The communication cost between cores is estimated by locality (local cores are assumed faster than remote cores).

2.4 Parallel and distributed SNN simulators

This section acts as an overview of how current state-of-the-art neuronal simulators implement parallel and distributed execution. It focuses on aspects of SNN simulations that are challenging for scalability, particularly around communication and workload allocation. Although the lens is on SNN simulations, most of the challenges are applicable to other complex system simulations with high degree of element interconnectivity.

There are three general steps or phases [97] when performing neuronal simulation: 1) solving phase, which involves the integration of differential equations in the neuron model; 2) contribution phase, where the propagation of messages (spikes) occurs; and 3) update phase, in which the changes to target neurons are applied. The solving phase is the easiest to parallelise [36] as it follows the SIMD model (a single function that needs to be applied to a large set of data). The propagation of spikes is the hardest and it becomes the general bottleneck on efficient parallel computation [79, 36], particularly as the size of the network grows.

There are three general approaches to parallelism in the literature of neuronal simulators, depending on the level of parallelism they target: task parallelism via distribution of work (using MPI), data parallelism by multi-threading (with the utilisation of GPU) and designing and implementing custom hardware (such as SpiNNaker [130], analogue circuits [199, 118] and FPGAs [43, 185, 50]).

Parallelisation via multithreading is the process of executing fragments of computation concurrently. It differs from distribution in that there is shared memory access and it generally subscribes to a SIMD model, were a single piece of code is applied to large amounts of data in parallel.

Current SNN simulators that offer data parallelism via multi-threading employ GPUs [186, 22, 228, 217, 208, 204, 75, 88] and multicore CPUs [35].

With the technological push of General Purpose computation on GPU (GPGPU), many research groups have turned to graphic accelerators to speed up their scientific

computing. Neuroscience has not been an exception and several simulators that take advantage of GPUs have been proposed: proof of concepts [208] (CUDA), [204] (OpenCL), and insights into suitable communication patterns and network representations [164, 36].

Other simulators based on code generation to produce kernels that can be used by GPUs include CARLsim3 [22], GeNN [228], ANNarchy [217], NeMo [75] and Neurokernel [139, 88]. These approaches optimise the generated code to enhance one or more of the following: parallelism (functions that can run concurrently), occupancy (usage of GPU cores), memory bandwidth (good memory access patterns), memory usage (via sparse representations) and thread divergence (avoiding threads of uneven execution time).

Although distribution on multiple GPUs is technically possible with some of those simulators, the programmer remains responsible for the distribution and management of the work.

Most sequential SNN simulators support *distributed execution* ([158, 97, 174, 171]) using the standard MPI to communicate between remote processes. They use the model based on [160] to distribute synaptic objects to be only represented in the node that contains the postsynaptic neuron. Thus when propagating spikes, only the index of the presynaptic neuron is required to be sent.

NEST uses a hybrid approach [174] combining distributed simulation across nodes -using MPI messaging- and multithreading within nodes with static, round robin-type allocation. Since NEST targets modelling of a large number of simple point-like neurons (not considering their morphology or location) simulations are normally communication-bound, i.e. when scaling the network size, the critical aspect is not the execution of neuronal equations (as these are relatively simple, particularly for parallel computation) but the propagation of spikes and synapse representation –especially in simulations with high communication frequency [124].

2.4.1 Communication challenge

Speed of runtime is important in general SNN simulations, for obvious reasons: running simulations faster means more experimentation iterations and less use of expensive resources, specially important in shared resources systems. Runtime is even more critical in particular cases such as when modelling Spike Time Dependent Plasticity, since behaviour of the network is ultimately dependent on the accurate timing between spikes and often experiments may require long simulation runs. Zenke et al. [229] show in a series of experiments with mid-sized networks that the bottleneck to speed gain in parallel simulations is ultimately due to latency in inter-process communication. Since this portion is no easily parallelisable due to interprocess dependencies [36], it becomes a key component to optimise. Eppler et al. [69] reports super-linear speed-up on small clusters and parallel systems (up to 10 processors) but then saturates at higher scales.

2.4.1.1 Propagation of messages

The propagation of spikes and its effect on target neurons in simulations is understood to be difficult to parallelise [134]. Brette et al. [36] identify this step as the bottleneck for GPU acceleration, as the operation does not follow the SIMD model. In simulations performed over analogue circuits where computation speed is much

higher, propagation of spikes becomes the practical bottleneck [199]. Neuron-based parallelisation [164, 163] and synaptic events-based parallelisation [75] approaches have been proposed to speed up the propagation of spikes in non-distributed systems. Distributed systems face an extra challenge: as the number of processes increases, the probability of a pre and post-synaptic neuron to be assigned to different processes increases, hence incurring in higher communication costs when spikes are propagated [124].

NEST [136] and *NEURON* [158] use a simple all-to-all communication pattern between processes to propagate spikes, implemented via MPI_Allgather function. Although it is simple to implement and generally does not impact performance in small parallel and distributed systems, its overhead quickly raises with the number of processes. Both acknowledge that the approach is a baseline and should be improved. *NCS 6* [106], hybrid distributed and parallel simulator, uses a similar approach whereby local parallel threads upcast the spike vector to the machine level and then all-to-all broadcast to share the information. Machines are then responsible to downcast the spike information down to threads.

A particular implementation of an MPI multicast communication pattern is proposed in [135] as an alternative to MPI all gather communication pattern designed on top of IBM's Blue Gene/P computer. It consists of Deep Computing Messaging Framework (DCMF) multicast and many-to-many efficient communication patterns that make use of the hardware architecture. However performance is lower when neurons have high number of connections (hence probability of any process to need to listen to all processes is high). In their proprietary simulator *HRLsim*, [159] uses non blocking point to point communications. However none of them show a comparison of the cost of communication and the gain between point to point and a collective message passing.

Other complex system simulators include publish-subscribe communication patterns [56] and reduce-scatter primitives to send messages only to processors that needed it [8].

SpiNNaker [130] follows an Event-Address Mapping scheme [123] to minimise network communication overhead between chips and boards. When a spike occurs, a standard message (or bundle of messages) is sent through a series of multicast routers, following each time a routing table until it reaches the target neuron. The routing table is pre-loaded upon initialisation with a planning algorithm that ensures that two neurons are connected using the shortest path between the chips in which they are hosted. The downside of this approach is that it is only directly applicable to simulations running on a hardware network (although MPI topology mapping can be understood as a soft routing table), and there are memory limitations to the information a routing table can efficiently store, particularly problematic in very large simulations.

2.4.1.2 Optimisation of message propagation

In a continuous SNN simulation run in distributed processes, after updating the states of neurons and synapses, each process needs to broadcast spike data so target postsynaptic neurons are notified of the activity of presynaptic neurons. This synchronisation event occurs at defined time intervals, with communication overhead at every step. One method of reducing the overhead is to **pack event-messages** together by using the intrinsic synaptic delay property of the neuronal

connection [160, 86]. The global minimum synaptic delay (smallest synaptic delay in the simulation) is used to set the frequency at which processes need to communicate. Hammarlund et al. [98] take this idea further and use a different frequency of communication per process pair, defined by the minimum synaptic delay in synapses across those processes, with a reduced overall communication overhead.

Most distributed SNN simulators use the Address Event Representation (AER) to **compress the spike data** exchanged between parallel processes. In software simulators, the AER approach by [160] is commonly used, where synapse objects are stored where the post-synaptic neuron is placed. Thus, processes containing firing neurons need only send the neuron ID and time stamp information to other processes. *HRLsim* [159] dynamically switches to an alternative bit representation [29] when the activity of the network is high. The bit representation describes the state (fired or not fired) of an ensemble of 32 neurons with a bit in a 4 bytes integer, allowing to represent the entire network more efficiently. Nageswaran et al. [164] targets the time stamp data to reduce the size of event messages. Instead of sending neuron id-timing pairs, they employ timing tables at which processes attach spike data, removing the need to include time data. Similarly, Morrison et al. [160] avoid having to send individual spiking timestamp by adding markers to the spike buffers (one per time interval between communication events), thus receiving processes are able to determine when the remote spike occurred.

After compressing and packing spiking data, parallel simulators must broadcast the messages to any and all processes that require it. Software simulators [42, 86, 106, 229] take the simple approach of sending every message to all other processes (*all gather*). Although this design scales poorly, it is easy to implement and for lower processes counts the hardware optimisations made for such a collective function call may hide this cost. To support the *all gather* approach, Lytton et al. [147] suggest that it may perform better than a discerning point-to-point, provided the number of processes involved is significantly lower than the average number of connections per neuron. In this situations, the probability that any neuron has post-synaptic targets in any other process is high, hence most spiking messages need to be sent to all other processes. In large scale simulations, however, with the order of thousands of processes, this may not be the case, as the average neuron connectivity will quickly be outnumbered by the number of processes, making all gather communication inefficient. The point at which average neuron connectivity outweighs the number of processes is dependent on the model in question; for human brain models it can reach tens of thousands, but for specific microcircuits it is significantly below that, such as the cortical microcircuit (3.8k, [177]) and the multi-area macaque visual cortex (5.8k, [195]). This relationship between average neuron connectivity and number of processes is behind the performance demonstrated by Kumar et al. [135] in their extension to *NEURON*, which showed higher gains in communication efficiency in low connectivity density models.

Previous works [158, 124] acknowledge that the communication overhead using *all gather* becomes a problem for scalable parallel simulations and **better broadcasting efficiency is needed**. *SPLIT* [98] and *HRLSim* [159] both implement point-to-point communication with reported gains over *all gather*. Jordan et al. [124] propose using *all to all* as an alternative collective communication in large scale *NEST* parallel simulations. Although they are more interested in reducing the memory footprint of communication at large scale, this collective method minimises

the redundancy of data sent across the simulation, with each process sending unique data to all other processes.

In large scale simulations performed to date, communication overhead is dealt with in several ways for each case. Kunkel et al. [137] and Jordan et al. [124] rely on message packing to reduce the communication overhead problem (communicate only every 15 time steps). In their custom cortical simulator Ananthanarayan et al. [9, 10, 8] use a point to point communication pattern to reduce communication volume. Their work includes strong scaling experiments showing that as the number of processors is increased, the simulation becomes communication bound. Other large scale simulations do not explicitly optimise communication as they do not study scalability [119, 68].

2.4.2 Workload balance challenges

Workload allocation in distributed and parallel systems is key to achieve high performance. The fundamental aspect for parallel applications is to divide workload in such a way that the time processes spend in useful computation is maximised. In other words, the overheads of parallelisation need to be reduced. Overheads arise from dividing the computation in independent processes, and include both extra communication and computation costs: sharing partial results (sending and receiving data), coordination (extra computation required to coordinate jobs between processes) and synchronisation (time processes are idle due to uneven workload assignment).

These costs increase with the number of distributed processes, thus the division of workload in independent portions and the assignment of those to physical computing nodes is key to HPC systems.

2.4.2.1 Allocation of neurons to processes in SNN simulations

Most SNN parallel simulators use round-robin allocation of neurons to computing nodes [86, 158, 171, 228]. This can be seen as a simple form of static load balancing as it tends to distribute evenly both number of neurons and incoming synaptic connections [137] —both of which contribute to overall workload cost. This results in almost perfect speed-up in compute-bound simulations, those in which most of the time is spent in the solving phase (updating neuron membrane potential and synapse objects), but with simpler artificial models the spike propagation is more predominant and hence the speed-up is reduced. Other extensions [105, 104] have gone further and split computation at the compartment level, but they are only applicable to multi-compartment neuron models. The importance of load balancing in neuronal simulations is evidenced by the efforts to include interfaces in *NEST* to load balancing algorithms[7].

Table 2.2 shows parallel and distributed SNN simulators and their load balancing approaches. Most of them use simple strategies to maintain workload balance such as round-robin or static allocation, with very few attempts to minimise communication between processes due to the connectivity of neuron elements.

Neurospaces [57] is one of the first efforts to explore systematic partitioning of neurons to processors. Neurons are represented as tree graphs that contain infor-

[7] `https://github.com/eth-cscs/nestmc-proto/issues/318`

mation regarding relative computational expense fed from manual tables. The algorithm traverses the graph and assigns neurons to partitions in round-robin. Once the partition workload reaches the expected balanced load it is finalised and mapped to a physical node.

NCS 6 [106] achieves workload balance with an arbitrary relative computational power ranking and cost estimate of neurons (by the number of incoming synapses). Neurons are sorted in decreasing cost and distributed across nodes, the one with lowest computational load (total cost over computational power) receiving the next neuron.

The approaches discussed focus on workload balancing to minimise simulation synchronisation or idle time. However, very little work has focused on the impact that neuron allocation has on communication (i.e. spike propagation). *HRLSim* [159] suggests assigning neurons based on how tightly connected they are but without details on how this is done. Urgese et al. [214] present an improvement to the default division of workload policy PACMAN [82] in SpiNNaker. In their work the authors use graph partitioning to minimise the amount of communication between groups of neurons placed in different partitions (i.e. chips and cores). Population models are broken down to produce a graph where nodes represent neurons and edges represent synapses. Spectral clustering is used to then group neurons into sub-populations, where tightly connected groups are kept in the same partition. The last step consist of mapping those resulting sub-populations into cores using Sammon mapping to adapt the multidimensional population space to a two-dimensional space representing the physical location of cores. If any core surpasses the maximum load allowed, an arbitration step moves sub-populations to other neighbouring cores.

A similar approach, but based on graph partitioning is employed in the complex system simulator *InsilicoSim*, now *PhysioDesigner*[8] [101, 102], demonstrating the potential benefits of graph partitioning to distribute computation. It then uses *METIS* [127], a popular graph partitioning package, to minimise communication cost between partitions (edge-cut) whilst maintaining balanced computation (workload between partitions).

Table 2.2: Overview of parallelisation and load balancing techniques on simulators

	Parallel		**Distributed**		**Load balancing**
	Multithreaded	**GPU**	**Multiple CPU**	**Multiple GPU**	
CARLSim 3	No	Yes	No	No	N/A
GeNN	No	Yes	No	No	Round-robin
Neurokernel	No	Yes	No	Yes	Manual allocation
Brian 2	Yes	Yes (with extension)	No	No	N/A
HRLSim	No	Yes	No	Yes	Connectivity tightness
NCS 6	Yes	Yes	Yes	Yes	Computational metrics
NEST	Yes	No	Yes	No	Round-robin
NEURON	Yes	No	Yes	No	Round-robin
PCSIM	Yes	No	Yes	No	Round-robin
NeMo	No	Yes	No	No	Round-robin
ANNarchy	Yes	Yes	No	No	N/A
SpiNNaker	Yes	N/A	Yes	N/A	PACMAN (static)
NeuroFlow	Yes	N/A	Yes	N/A	Based on spatial proximity

[8] `http://physiodesigner.org/`

2.4.2.2 Workload distribution in other parallel applications

The application of dynamic scheduling algorithms to complex system simulations is common in agent-based modelling (ABM) parallel systems. They offer life-cycle management, inter-agent communication, agent perception and environment management [187].

Traditional approaches to dynamic load balancing focuses on even spread of computational load with regular re-evaluations throughout runtime. Chow et al. [51] propose a composite formula to calculate agent load that incorporates inter-agent communication cost for their risk detector ABM system *Comet*. Results show that workload is more balanced amongst distributed machines with their metrics than when considering computational load alone.

D-Mason [55] and *Pandora* [188] use a partitioning of agents done via an agent area of interest (AOI) (positional partitioning). In runtime, agents are allowed to migrate to neighbouring grids, mimicking their physically movement. It uses a resource-centric approach to allocate agent-to-agent interaction work, as *workers* (physical cores) register their willingness to compute overlapping AOI based on their computational capacity.

Other scientific simulations can take advantage of the spatial features of the objects to be simulated, i.e. they can be naturally sorted in a location coordinate system. Hence, those fields make use of domain decomposition [94] techniques to break down the real space into a grid to parallelise the solution of the equations. In engineering many parallel linear iterative solvers for differential equations [66] have been proposed [162, 179, 38].

NWChem [215] is a framework for large-scale chemical simulations. It provides functionality to manually achieve data and task parallelism via a custom toolkit, with a distributed shared-memory model where data transfers (from global address space to local storage) and message passing between clusters using MPI are handled by the user.

2.5 Graph partitioning for workload distribution

Agent-Based Simulations (ABS) are a broad class of models that can be used to represent and simulate complex systems based on simple single-agent rules and messages to describe interactions between those agents. ABS have three general phases: update, where the state of each agent is recalculated based on the previous state; communication between agents, which is handled via discrete messages; and consolidate, where the information on the messages received by an agent is incorporated into its state. SNN simulations can be thought of as a type of ABS in which neurons and synapses are modelled as agents that communicate discrete messages amongst themselves (i.e. spikes). The computation consists of updating the internal state of the neurons and synapses.

The computation and consolidation phases of an ABS are an ideal fit for distributed computing as agents can be updated in parallel, achieving a speedup proportional to the number of computing nodes. The communication phase, on the other hand, is notoriously hard to parallelise. When two agents that need to share data are allocated in different computing nodes, a message needs to be sent across the network. With more computing nodes, the probability of agents that require

communication being hosted in different nodes increases and with it so does the required network communication.

To attempt tackling the scaling problem, ABS can be modelled as a graph [11] where vertices represent agents and edges represent connections between those agents (potential communication channels). Once the system is represented as a graph, optimisation algorithms can be applied to map computation (vertices in the graph) to computing nodes. Graph partitioning is particularly suitable as it can optimise multiple criteria such as workload balance (number and weight of agents assigned to each partition) and communication (edges that go across partitions). Graph partitioning has been applied in the past for load balancing purposes in SNN domain [57, 214] and other complex computations and simulations [166, 101, 102, 27].

Although graph partitioning has been applied to optimise ABS, the growing challenges presented by large-scale distributed computing increase the complexity of the optimisation problem which remains unsolved.

2.6 Graph partitioning problem

Graph partitioning can be used to find solutions to workload distribution to optimise both workload balance and communication overhead. The graph partitioning problem can informally be described as a procedure to divide a graph of interconnected nodes into balanced partitions to minimise the weight of the edges cut by partitions. Balancing aims to evenly distribute the weight of the vertices associated with each partition, but standard strategies can accommodate multiple criteria (i.e. multiple weights per vertex, weighted edges). This is particularly interesting for multiphase simulations where optimising for one parameter may imbalance execution time for other phases.

Formally, a graph $G = (V, E)$ consists of V, a set of weighted vertices and E, a set of weighted edges, where each edge is a pair of vertices $e = \{v, u\}, v \in V, u \in V$. Given the graph G, and a positive integer p, the graph partitioning problem involves finding p subsets of V such that $\cup_{i=1}^{p} V_i = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The goal is to minimise the edge-cut, the sum of the edge weights crossing from one partition to another. For a *balanced* graph partition, an extra constraint is added $W(i) \approx W/p$ for $i = 1, 2..., p$, where $W(i)$ is the sum of all node weights in partition i and W is the sum of all weights.

Graph concepts:

- **Edge cut**: number of edges that connect nodes in different partitions or clusters. It is traditionally the metric used to assess quality of partition (low best).

- **Vertex Degree**: for a given vertex, its degree is the number of incident edges (incoming connections from other vertices). This is an indication of how interconnected a vertex is, whether it is a hub. Vertex with high degree are likely going to be associated with high costs of communication.

- **Power-law** or sale-free graphs: a scale-free graph is one in which the number of edges initiating from a given vertex follows a power-law distribution (probability of vertex to have degree d is $P(d) \approx d^{-\alpha}$, with α controlling the skewness). Intuitively this results in graphs with a few vertices connected

to a large proportion of the population. Power-law graphs are particularly challenging to partitioning as it is shown in section 2.8.1.

- **Hop cut**: inter-partition edges (edges cut) count weighed by the physical cost of communication.

2.6.1 Graph partitioning in distributed simulations

Graph partitioning can consider the cost to optimise neuron to processor mapping [115, 219, 161]. More broadly, the mapping problem is central to any multiprocessor application in general, and specifically to all distributed complex system simulations.

In the context of distributed agent based model simulations, [11] analyses the effectiveness of graph partitioning to reduce communication costs, and highlights a strong negative correlation between simulation time and the edge cut of the graph. Brette [36] identifies graph partitioning as a potential tool to minimise the amount of communication overhead by grouping spike messages together. SNN simulators that support distributed execution generally use some type of partitioning to divide workload, but it is limited to round-robin or simple workload balancing approaches —see section 2.4.2.1. SNN simulators do not formally utilise graph partitioning to divide the work into nodes, but authors have pointed out its applicability [214]. Graph partitioning can be used to divide the workload based on parameters such as network connectivity, computational power and communication speed.

Users of SNN simulations frequently describe their targeted models at a population level, with a list of neuronal populations and a deterministic or probabilistic connectivity between them to define the system. Figure 2.3A shows a simplified Cortical Microcircuit model at this level. Populations are eventually broken down to the neuronal level by SNN simulators and then mapped to computing nodes. To be able to use graph partitioning to aid workload allocation, these models need to be represented as a graph of vertices and edges. A direct translation is to represent neurons as vertices and synaptic connections as edges. Figure 2.3B shows the graph representation that corresponds to the Cortical Microcircuit model. An advantage of using graph to model SNN is that one can modify the level of granularity of the unit of workload, i.e. instead of having vertices for neurons, vertices can represent higher level concepts such as sub-populations or regions.

Figure 2.4 shows different graph types, from independent networks, to small world and random connectivity. Biological plausible SNN are expected to sit somewhere in the middle of this continuum [39], where locality is relevant, i.e. local connectivity is expected to outnumber remote connectivity. This is also the case for physically based (location) complex system simulations such as traffic, chemical or particle simulations, due to the fact that interactions mostly happen locally, i.e. between nearby agents. This suggests that using connectivity-agnostic workload allocation algorithms such as round-robin may incur higher communication costs due to increased inter-process connectivity —connected neurons hosted in different partitions. Graph partitioning can be used to find local connectivity hubs and therefore improve simulation performance, as Chapter 3 demonstrates.

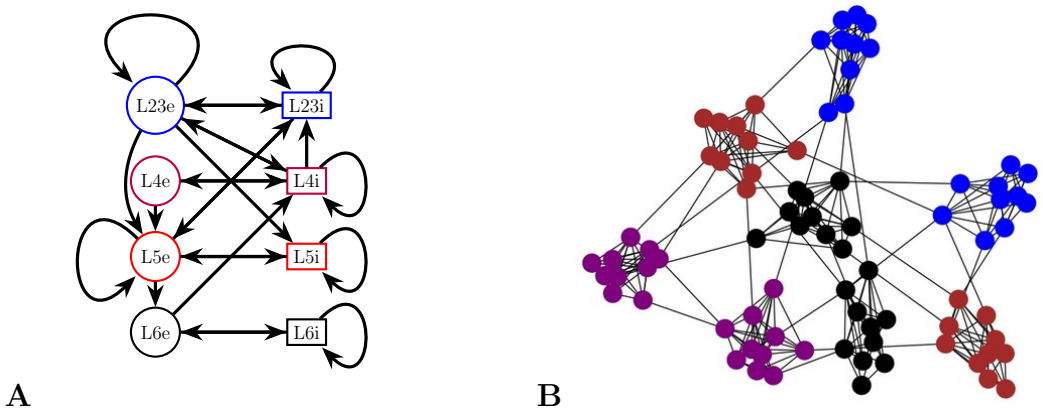


Figure 2.3: Cortical Microcircuit neuronal model (only connections with p > 0.05 shown).**A**: High level definition of the CM model with excitatory (circle) and inhibitory (rectangle) populations. **B**: Simplification of a low level representation of the same CM model at the neuron (circle) level.

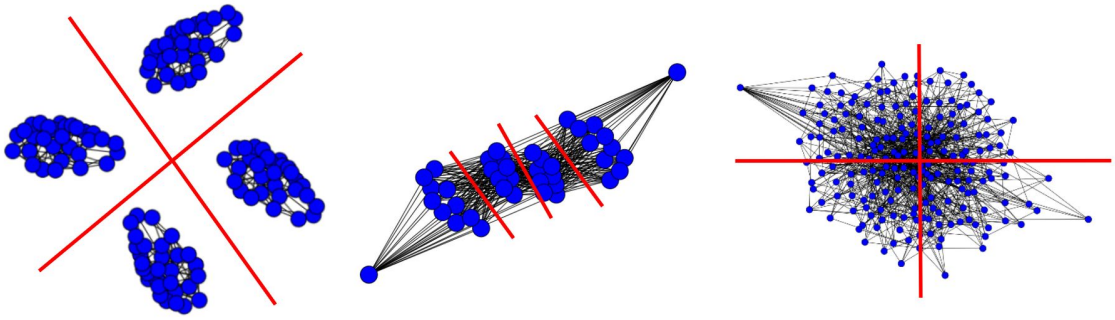


Figure 2.4: Graphs to model SNN mapping to computing nodes. From left to right: isolated networks, small world networks, random networks. Biologically plausible SNNs lie somewhere in the middle, which gives the opportunity to optimise neuron allocation by placing highly connected neurons in near computing nodes.

2.6.2 Graph partitioning algorithms

Graph and hypergraph partitioning is an NP-Complete problem [83], which means one must use heuristic approaches to find approximate solutions. For more in-depth surveys on the field please see [165, 200], or [40] for recent advances. Here we highlight the most commonly used algorithms.

Graph partitioning algorithms can be classified into three groups [165] depending on what they prioritise:

- **Local improvement** methods: based on local search, they process a vertex at a time to optimise the cost (usually edge-cut). Local methods usually have less memory requirements as they do not need to work with the entire graph concurrently, but tend to find local minima.

- **Global methods**: they work with the entire graph to produce a solution, i.e. set of partitions. Heavier in memory requirements, they tend to provide higher quality results since they have full knowledge of the graph.

- **Multilevel algorithms**: the graph is coarsened prior to partitioning, and then it is expanded to assign nodes to partitions. Multilevel algorithms are a compromise between local and global methods, providing good quality results with a fraction of the memory requirements of global methods. Current state of the art partitioning methods use multilevel algorithms.

Classic local improvement methods are based on Kerningham-Lin (KL) algorithm [129]. KL iteratively attempts to improve 2 given partitions by finding an approximation to the best exchange of λ pairs (any number of nodes from one set to the other) that minimises the overall edge-cut. Fiduccia and Mattheyses (FM) [76] proposed an improvement on KL to tackle the exponential costs of selecting a candidate to exchange and having to recalculate gains when nodes are swapped. Although KL and FM operate on 2 partitions, they can be easily extended to be k-way by operating recursively. Other search-based algorithms have been presented: memetic algorithm with tabu search [81] and simulated annealing [180].

Popular global methods are based on:

- Distance-based methods generally involve selecting two nodes within the graph to act as centroids for two future partitions [200], then sort all nodes by their relative distance. The distance list is finally used to assign nodes to partitions. Various algorithms vary in their distance measurement: Euclidean distance (Recursive Coordinate Bisection); longest path distance (Recursive Graph Bisection); eigenvector of the Laplacian matrix [176] (Recursive Spectral Partitioning).

- Growing methods start with a random node and traverse the graph using breadth first to keep adding nodes until more than half are assigned to the cluster. The remaining are assigned to the other cluster. Often requires several restarts to find good partitions.

Multilevel graph partitioning [103] is a three step process that consists of: 1) *coarsening* or clustering of the original graph; 2) *partition* performed on the coarse graph; 3) final expansion or *uncoarsening* to assign vertices to partitions with a final local improvement to refine them. Various versions exists based on the coarsening algorithm chosen, although the most common one is contracting a pair of nodes (single edge). The edge selected is determined using edge rating functions, which trade off between edge weight and node weight uniformity. Multilevel partitioning is usually a very effective heuristic for several reasons: at coarse level, partitioning can be much deeper (more work per step) since there are less elements involved; larger exploration of the solution space since single coarse node move produces big changes to final solution; final local improvements are expected to run fast as they start from a good partition already.

Most modern graph partitioning algorithms are based on multilevel approach [127, 219, 3, 211, 138, 115, 48, 31].

2.7 Hypergraph partitioning for workload distribution

Hypergraphs are a mathematical generalisation of graphs in which edges can connect any number of vertices (thus are called hyperedges). Graphs are good at

representing computing elements and their interactions. However, hypergraphs have been shown as better models of communication in parallel and distributed applications [63, 61] as they can represent communication groups naturally. Hypergraphs are relevant in a wide variety of domains: applications in VLSI [140], social network analysis [122], distributed database design [125, 226], text [114] and image [144] retrieval and machine learning [221].

2.7.1 Hypergraph formalisation

A hypergraph $H = (V, E)$ consists of a set of vertices V and a set of hyperedges E, where each hyperedge is a subset of V that defines the connectivity pattern. The size of each hyperedge is denoted as its *cardinality*. Hypergraphs are a generalisation of graphs that can have any cardinality, i.e., one hyperedge connects multiple vertices, where graphs have a maximum cardinality of 2. Hypergraph partitioning is a process that assigns vertices to partitions in such a way that a connectivity metric (usually hyperedge cuts, or hyperedges that span more than one partition) is minimised. To avoid trivial solutions that minimise the hyperedge cut (such as assigning all vertices to one partition) partitioning algorithms maintain load balancing by only allowing solutions that have a total imbalance factor that is below a specified value. The total imbalance I is calculated dividing the maximum imbalanced partition in the scheme by the average imbalance across partitions. Formally:

$$I = \frac{\max_{p \in P}(L(p))}{(\sum_{i=0}^{|P|} L(p_i))/|P|}$$

where P is the set of partitions and $L(p)$ is the load cost for partition p defined as the sum of the weights of all its nodes, $L(p) = \sum_{i=0}^{N} W(n_i)$ where N is the number of nodes in partition p and $n_i \in p$. The total imbalance must be lower or equal than an arbitrary tolerance value.

Hypergraphs are good at modelling parallel communication when each hyperedge represents a frequent communication group of vertices. The higher external degree per hyperedge (more incident partitions, formally defined below) the more the modelled application will have to send data across partitions and hence more communication is required. In chapter 3 I demonstrate that hypergraphs can be used to model large scale distributed scientific simulations [73]. When using hypergraphs to model parallel applications, the goal is to partition the hypergraph in k partitions, where each partition represents a computing node in the hardware architecture the application runs on.

Hypergraph-specific concepts:

- **hyperedge cut**: number of hyperedges that contain vertices allocated to more than one partition or cluster.
- **Hyperedge degree** or **cardinality**: number of participating vertices in a hyperedge. When using hyperedges to represent vertex communication groups, the cardinality is an important metric to determine the cost of cutting a hyperedge.
- **Connectivity metric** or **fanout**: sum of all hyperedges cut, weighed by the number of participant partitions minus one. This is an alternative objective

metric to hyperedge cut to minimise that closely models total volume of communication in parallel applications.

- **Sum of External Degrees** (SOED): an external degree of a partition is defined as the number of hyperedges that are incident in it and also in at least another partition. The SOED is the sum of all external degrees across partitions.

2.7.2 Hypergraph partitioning algorithms

Partitioning algorithms for hypergraphs with good quality results have been proposed, using a variety of techniques: multilevel partitioning (*PaToH* [45], *hMetis* [128], *ParKway* [211]), multiconstraint [14, 61] and with multilevel with fixed vertices (*Zoltan* [31]). Parallel versions, which allow the algorithm to tackle larger hypergraphs faster, also exist [62, 210].

2.8 Challenges to graph and hypergraph partitioning

There are three general challenges to graph and hypergraph partitioning: limitations of current algorithms on large-scale graphs; partitioning quality on power-law natural graphs; and modelling heterogeneity. The first two are discussed in this section, and the latter issue is described in section 2.9.

2.8.1 Highly skewed and power-law graphs

In previous sections we have discussed graph and hypergraph partitioning as the allocation of vertices to partitions. This is specifically referred to as vertex partitioning. An alternative approach is to allocate edges instead. This has been shown to be superior to vertex partitioning in certain domains such as partitioning power-law graphs [189, 90], graph processing [122] and distributed database entity placement [125, 227].

Intuitively, the task in edge partitioning is analogue to that of vertex partitioning but assigning edges to partitions instead of vertices. When an edge is assigned, it results in the partition holding both associated vertices (or more for hyperedges) in it. Because multiple edges can include the same vertex, this results in *replicas* of that vertex spanning across all partitions that contain edges incident to that vertex —see right diagram in Figure 2.5, where all vertex replicas have been coloured in red.

Given the graph $G = (V, E)$, with V vertices and E edges, and a positive integer p, the edge partitioning problem consists of finding p subsets of E such that $\cup_{i=1}^{p} E_i = E$ and $E_i \cap E_j = \emptyset$ for $i \neq j$. The vertex-cut is defined as the set of vertices that span across more than one partition. The vertex-cut is also referred to as the *vertex replication factor* (VRF). The common target in edge partitioning is to minimise the vertex replication factor.

Although vertex partitioning approaches to tackle highly skewed vertex degree distributions graphs exist [3], it has been shown that edge partitioning approaches

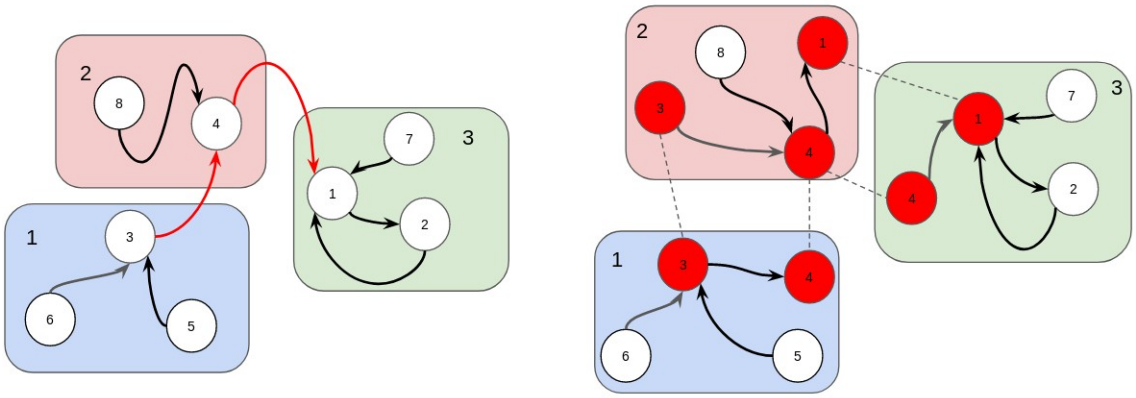


Figure 2.5: Edge vs Vertex partitioning diagram. Both diagrams show the same graph connectivity. Left: vertex partitioning (highlighted in red the edges that are being cut). Right: Edge partitioning (highlighted in red the vertices that are replicated).

are far more suitable for this type of graphs and hypergraphs [189, 90]. PowerGraph [90] proposes a graph processing abstraction designed to work with power-law graphs, offering an edge partitioning placement to distribute the graph in several machines. It uses a greedy approach to sequentially assign edges to partitions to reduce vertex replication.

HDRF [173] builds upon PowerGraph but instead of using a simple greedy assignment of edges it uses an objective function that includes the partial degree of vertices involved in the edge (an approximation of the actual vertex degree that is built from past observations), the current vertex replication factor in the partition and a load balancing parameter. Sajjad et al. [189] offer a parallel implementation of HDRF with a centralised state to share the partial vertex degree amongst otherwise independent substreams of edges, resulting in significant speedup.

Combined vertex-edge partitioning algorithms have also been proposed (*HYPE* [122]) to attempt to balance the benefits of either method. *HYPE* uses iterative label propagation. In each iteration, a hyperedge is assigned a label based on the label (i.e. partitions) of its incident vertices. The optimal label is found based on a objective function that takes balance and reduction of replicas into account. The same is done per vertex, considering the labels of all its hyperedges.

Edge partitioning based on minimising vertex replication factor lends itself naturally as an ideal partitioning algorithm to distributed databases [227]. When distributing data across different machines, query cost is directly related to where the data entities are hosted, hence entity copies are situated (and replicated) where they are needed. Data requests have to be generated to communicate updates when the entities change, with communication increasing with the number of replicas. Edge partitioning has been used to reduce the number of replications [125].

2.8.2 Limitations to large-scale hypergraph partitioning

Large hypergraphs are increasingly relevant in research and applications (social media graph analytics, web networks, recommendation systems). As the size of the graphs increases, it surpasses the local resources of a single machine, both in terms

of memory (too big to fit in memory) and computational capacity (too large to be processed in an acceptable time scale). Parallel multilevel partitioners alleviate some of the burden by distributing the coarsen hypergraph [31]. However, they struggle to tackle very large graphs [49, 125] because the coarsening phase and refinement during uncoarsening do not scale well.

Since converting a hypergraph to a graph cannot be done without compromises (eg. size of the graph) [117, 122], new hypergraph partitioning heuristics have been proposed to tackle large-scale hypergraphs. The most common heuristic is based on local swapping or refinement of an existing partitioning, where vertices or edges are swapped between partitions to maximise the gain measured by an objective function.

The Social Hashing Partitioner (*SHP*) [125] proposes a vertex partitioning algorithm to reduce storage sharding in distributed databases. Storage sharding occurs in database queries when records that are linked are hosted in different machines, therefore requiring messaging to fulfil the request. The algorithm optimises for minimum hyperedge fanout by performing local search on all vertices. The local search is done distributively to allow for large hypergraphs.

HyperSwap [227] is another local swapping approach that uses edge partitioning for scalable distributed hypergraph partitioning. In contrast to *SHP* it performs swapping of hyperedges instead of vertices. [226] is similar but with a centralised heuristic to improve quality and partitioning performance.

Local search can often fall into local optima. To increase the quality of the partitioning, Nicoara et al. [166] uses *METIS* as a starting point for the local search and refines it with distributed vertex migration to improve edgecut whilst maintaining load balance. Although using a better starting partitioning does improve the final quality, using a static partitioner such as *METIS* acts as a practical bottleneck for the size of graphs to be used: *METIS* is a sequential algorithm and does not scale well.

In addition to local swapping, heuristics to optimise hypergraph processing have been proposed. *HyperX* [122] is a hypergraph processing framework with a novel partitioning algorithm: label propagation partitioning (*LPP*). *LPP* performs iterative partition update using hyperedge and vertex labelling, guaranteeing that both vertex and hyperedges are balanced. In each iteration, a hyperedge is assigned a label based on the label (partitions) of its incident vertices (optimal one is found based on a objective function that takes balance and reduction of replicas into account). The same is done per vertex, considering the labels of all its hyperedges.

In Chapter 6 I explore the memory limitations on large scale hypergraphs for state-of-the-art global partitioners [31] and how streaming and restreaming algorithms are more suitable for such tasks -see section 2.8.3.

2.8.3 Dynamic and unknown graphs

Large graphs and hypergraphs may require more memory than is available to computing nodes. This is particularly true in the era of social media graphs, where real applications must handle billions of nodes and trillions of edges ([122]). Due to increasing memory requirements, hypergraphs may not be knowable all at once centrally. Furthermore, large hypergraphs in real applications are likely to be dynamic with regular changes to nodes and hyperedges. Those limitations are not handled

adequately by state-of-the-art global partitioning algorithms. Streaming algorithms are designed to tackle those limitations to produce partitionings in challenging large scale graphs.

Streaming graph partitioners differ from global ones (such as multilevel or recursive bisection) in that vertex allocation decisions are made based on local, partial information. This means the algorithm does not have the entire graph in view when calculating the cost of allocating a vertex to a partition. They are frequently called greedy since once they make a decision, it is not revoked later on after seeing more vertices.

When an algorithm applies more than one pass (repeats the stream that visits the vertices once), it is often referred to as a restreaming approach. When performing restreaming, i.e. reassignment of units to partitions, it is important to consider the cost of migrating data [46, 47] against the gains brought by the new streaming process.

FENNEL [212] is a generalisation framework for streaming partitioning. Its objective function includes the cost of edge-cut and a balancing parameter (i.e. number of vertices within a partition). Performance is shown to be competitive against offline methods (e.g. *METIS*) in certain scenarios, and generally more scalable. *FENNEL* shows that Linear Deterministic Greedy is one of the most effective algorithms, where each node is assigned to a partition based on whether any of its neighbours is already assigned to it and the remaining capacity of the partition.

Streaming partitioning prioritises low partitioning latency over partitioning quality, with the risk of finding local optima. *ADWISE* [153] proposes a controllable algorithm that allows to find an adequate balance between the two for the problem at hand. It does so by considering a window of edges at a time, instead of one (for streaming) or all (for global). The window size is adaptable and found automatically.

HYPE [152] performs hypergraph partitioning using heuristic neighbourhood expansion. Neighbourhood expansion has been formulated for graph partitioning [230], but up to now was unfeasible for hypergraphs due to the large size of neighbourhood set for each vertex in a hypergraph. Partitions are grown via neighbourhood expansion, one vertex at a time, in a form of streaming. Instead of arbitrarily receiving vertices, the algorithm grows by selecting neighbouring vertices.

2.9 Communication heterogeneity challenge

Previous work has already highlighted the impact that uneven computation and communication architectures in HPC and Cloud computing has on computation performance [223, 225, 48]. The communication heterogeneity is significant enough to be considered in the workload allocation of distributed applications. Although there are graph partitioning approaches that are architecture-aware, this is not the case for hypergraph models. This section surveys the current literature in architecture-aware partitioning as a means to optimise workload allocation in heterogeneous distributed systems such as HPC and Cloud clusters.

2.9.1 Architecture-aware graph partitioning

In order to factor architecture heterogeneity in both communication and computation, graph partitioning algorithms frequently build a model of the target hardware

in the form of a graph. This graph, commonly referred to as **machine** or **physical graph**, models computation as weighed vertices and communication bandwidth or cost as weighed edges. To help differentiate between the machine graph and the target graph to partition, the latter is referred to as **application** or **data graph**.

Authors have indicated that edge-cut weight alone is an inadequate measurement when mapping machine graphs in heterogeneous architectures [219, 233]. In topologies with uneven communication between nodes (bandwidth and latency) optimising for edge-cut may lead to worse performance by using slow channels of communication. Hence there is a need for architecture-aware partitioning that goes beyond computational load.

Walshaw and Cross [219] consider communication links between processors as part of the partitioning algorithm. They use a multilevel algorithm with a cost function that combines edge-cut with the weight of the processors connection.

Another aspect in which partitioning algorithms can be architecture-aware is by considering the availability of memory and computational resources. It has been shown that suboptimal execution times are found [233] if local resource contention is not considered. A well-known case is the impact of data locality and memory access patterns in performance in GPU parallelism [64, 167]. Unaware algorithms place as much communication and computation as possible within one node as the cost associated during graph partitioning is zero. This situation could lead to high computation and communication in neighbouring cores, increasing the probability of resource queuing to wait for availability. *PARAGON* [234] and *PLANAR* [232] are parallel partitioning refinement algorithms that take resource contention and network heterogeneity into consideration. Their optimisation goal is to minimise the edge-cut whilst reducing the edge-cuts amongst partitions with high communication costs (known as *hop-cut*). To consider resource contention they use a penalty score based on the degree of contentiousness between cores. A similar score is used for load balancing in [131].

Teresco et al. [205] present a hierarchical partitioning algorithm that takes into consideration the communication cost of the network. Although the original library that provided the functionality has been discontinued, the hierarchical partitioning feature is still part of *Zoltan* as a system for resource-aware partitioning using a tree like structure to model the communication and processing power of the network.

In the context of job scheduling, Xu et al. [223] demonstrate an adaptive streaming graph partitioning algorithm able to deal with heterogeneous networks, both in terms of computational capability and communication cost. It receives one vertex at a time and must decide to which partition it belongs from the info it already has. The objective function contemplates computational capabilities of each node as well as relative communication costs. Thus, the target is to reduce job completion time rather than exclusively minimising edge-cut.

Previous attempts to achieve architecture-aware graph partitioning have been proposed to multiple partitioning strategies: local refinement, streaming and global approach. They are discussed below.

Local refinement refer to strategies that take the current graph partition and improve it by swapping vertices around. *PARAGON* [234] is a parallel refinement partitioning that considers physical communication cost and resource contention to move vertices between partitions. Moulitsas and Karpys [161] present a refinement approach to graph partitioning that includes architecture-specific information to

improve partitions. The initial partitioning optimises computational balance and memory resources amongst processors. The refinement stage considers the machine graph to modify initial partitioning. A greedy algorithm (swap vertices) is employed to minimise communication volume.

Streaming partitioning: [225] proposes graph streaming over a machine graph in which vertices represent computing nodes and edges represent communication bandwidth, both weighed to reflect the system capabilities. Vertices from the application graph are then assigned using a greedy function that balances the weighed computation and communication. [154] uses edge streaming greedy policy to reduce vertex replication, then clustering to partition the machine graph and map partitions to physical clusters, considering heterogeneity.

Global approach: *Surfer* [48] is sequential network aware graph partitioning for graph processing in the cloud. The authors recognises the heterogeneity of cloud computing bandwidth as a consequence of tree topology connectivity infrastructure. Surfer models the network as an undirected graph that is recursively partitioned (multilevel algorithm) synchronously with the data graph until the data graph partitions can fit in main memory.

Partitioning algorithms for hypergraphs with good quality results exist using a variety of algorithms: multilevel partitioning (PaToH [45], hMetis [128], ParKway [211]) and multiconstraint [14, 61]. Unfortunately none of those approaches considers the physical architecture of the network. When modelling parallel applications as a hypergraph, not only it is important to reduce the hyperedge cut, but also the hop cut (connections including the physical cost of communication).

Zoltan [62] offers a hierarchical approach for partitioning a hypergraph. It allows users to partition their hypergraphs at different levels of granularity, using a sequence of partitioning schemas (refinements on subgraphs). Each level can be used to model a level in the architecture hierarchy (socket, board, group, cluster). The focus of the approach is on being able to use high cost algorithms at levels where reducing communication is more important and low cost ones when the communication may not impact as much. However, this approach only establishes qualitatively differences between architecture levels (whether a pair of nodes connection is faster or slower than another pair) and does not model well the cost of communication between computing units belonging to different hierarchies (how much faster or slower).

2.9.2 Other approaches to model communication heterogeneity

In section 2.3.3 we described topology mapping. By modelling the machine graph and the application graphs topology mapping can be used to optimise the assignment of workload to computing nodes at a high level of granularity [112, 184, 181, 209, 23, 157, 121].

Dynamic load balancers are focused on maintaining workload balance to reduce idle time between processes. Although they are generally not concerned with communication costs (let alone variable costs), some approaches do model it to optimise simulation runtime. Bhatele et al. [24] proposes a centralised approach in which a process collects compute and communication metrics of compute elements in molecular simulations. That information is used to make decisions about load balancing

and to place communicating objects in nearby computing nodes. The simulation space is divided using geometric decomposition to group atoms in patches of equal size, then they are mapped to the hardware, keeping load balance and minimising communication by avoiding placing communicating objects in nodes that do not have other connected objects (thus avoiding creating more communication). The compute is modelled as a different object which is mapped to nearby nodes, making the approach topology-aware.

Dynamic schedulers target task or job-based allocation in distributed machines. Although there may be some inter-task communication, the main concern in this scenario is task dependency and job execution time. Thus, dynamic load balance systems are more concern with computation heterogeneity [13, 16, 26, 33, 32, 52, 84, 183] and not communication heterogeneity.

2.10 Identified areas of research

Based on the review of the literature in this chapter, there are several areas of research that have been identified. The remainder of the thesis is structured to target each of these areas:

- Hypergraphs are good models of communication in parallel applications [63, 61]. Can hypergraphs be used to model and optimise communication in SNN simulations? (Chapter 3).

- MPI collectives do not scale well [206], particularly in distributed applications that exhibit sparse communication patterns [111]. Can hypergraph partitioning drive sparsity to make dynamic sparse communication a more efficient collective for SNN simulations? (Chapter 3).

- Communication heterogeneity in HPC has a significant impact on performance [223, 225, 48], however hypergraph partitioning algorithms do not consider this heterogeneity when performing partitioning. This might impact the performance of distributed applications that are modelled as a hypergraph. Are there performance improvements, measured in a reduction of communication time at runtime, to be gained by incorporating bandwidth data to the partitioning process? (Chapter 4)

- Hypergraphs models are useful in many applications, such as VLSI [140], social network analysis [122], distributed database design [125, 226] and distributed and parallel applications [63, 61], which highlights the importance to have appropriate benchmarks and datasets to test and evaluate hypergraph partitioning strategies. However, current benchmarks [193] focus on static partitioning quality metrics (hyperedge cut, SOED) which are of limited use when using hypergraphs to model dynamic applications such as distributed simulations. More adequate benchmarks (Chapter 4) and datasets (Chapter 5) are needed to accurately measure the impact that hypergraph partitioning has on the performance of distributed simulations.

- Current state-of-the-art hypergraph partitionings use a multilevel approach, achieving a good balance between lowering memory requirements and reaching good quality results. However, they struggle to tackle very large graphs

[49, 125] because the coarsening phase and refinement during uncoarsening do not scale well. Streaming partitionings can deal with dynamic, not-fully known graphs. To date there have been very limited efforts dedicated to streaming hypergraph partitioning. Can efficient, scalable, streaming hypergraph partitioning be used to model distributed communication and map its communication patterns to the hardware bandwidth specifications it runs on? (Chapter 6)

Chapter 3

Communication sparsity in distributed Spiking Neural Network Simulations to improve scalability

3.1 Overview and contributions

Chapter 2 evidences the limitations of state-of-the-art SNN simulations at large-scale, where distribution and parallelism of computation is required:

- Common distribution of workload algorithms (round robin, random and manual allocation) do not take into consideration connectivity locality, which is natural in biological networks ([39, 170]). This may lead to increased communication requirements when distributing the simulation across multiple computing nodes.
- Hypergraphs have been presented as good models of communication in parallel applications [63, 61], but their applicability to SNN is not well understood.
- State-of-the-art SNN simulations use dense communication collectives to distribute spike data. These collectives have been shown to not scale well [206]. Sparse communication collectives have not been explored and they have been suggested to incur in lower overheads when the application's pattern of communication is sparse [111].

Research question: Can the sparsity of connectivity inherent in biologically plausible SNNs be exploited to improve the efficiency of collective communication methods and reduce simulation time?

This chapter tackles both limitations to demonstrate that sparsity in distributed SNN simulations can be exploited to improve simulation runtime by reducing overheads of communication. The contributions of this chapter to the overall thesis are:

- **C1**: (Chapter 3) Demonstrate that communication sparsity between computing nodes drives performance in communication-bound SNN simulations.

- **C2**: (Chapter 3) Produce sparser communication patterns (up to 90% less Average Runtime Neighbours and up to 80% less volume of data) by modelling SNN simulations as a hypergraph and using partitioning algorithms.

- **C3**: (Chapter 3) Reduce the overheads on the three phases of P2P communication (synchronisation, handshake and data exchange) in SNN simulations with the use of dynamic sparse communication patterns, resulting in more balanced inter-process communication (up to 90% less implicit synchronisation time), faster simulation runtime (up to 73% less time) and more computational efficiency (up to 40.8 percentage points more time spent in computing).

To facilitate the experimentation and evaluation of alternative workload allocation and communication strategies, a novel framework for testing communication and workload allocation strategies Spiking Neural Network simulations is developed (contribution **C4**).

3.2 Communication in timestep-driven simmulations

Simulations of Spiking Neuronal Networks consist of discrete time-steps at which the membrane potential of the neurons involved is recalculated. When the potential reaches a certain threshold, neurons fire, producing a *spike* that is transmitted to connecting neurons. This spike in turn affects the membrane potential of receiving neurons. Thus, the simulation is divided in two phases: *computation phase*, at which neuron and synaptic models are updated based on partial differential equations; and *exchange phase*, where the spikes are propagated from firing neurons to post-synaptic targets.

3.2.1 Phases in P2P communication

In distributed systems, the exchange phase involves inter-process communication when pre- and post-synaptic neurons are hosted in different processes. In such cases, inter-process dependencies are introduced that force them to be *implicitly synchronised* (at the same step in the simulation) before starting *data exchange* (sending and receiving spiking information).

State-of-the-art SNN distributed simulators have demonstrated the effectiveness of the use of point-to-point (P2P) communications [159, 98, 124, 135]. Using P2P introduces a necessary handshake protocol between processes to know which processes each one has to listen to. Thus, each communication stage consists of three phases (Figure 3.1):

1. Implicit synchronisation, in which processes must wait until every other process is ready for communication. Computation and communication imbalance contribute to this phase.

2. Communication handshake (informing target processes of the intent to communicate). In naive P2P communication this operation has $\mathcal{O}(P)$ complexity [111], with P number of processes, a problem for scalability.

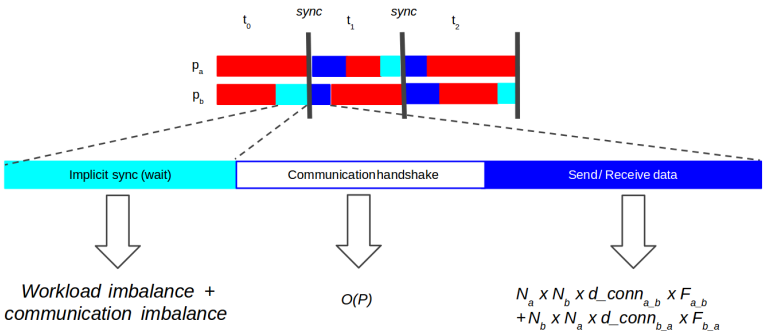


Figure 3.1: Diagram showing synchronisation phases in point-to-point (P2P) communication. **Implicit synchronisation** (light blue): processes waiting for each other to start communication. **Communication Handshake** (white): explicit notification to other processes of the intention to send data (receiver processes must prepare buffers). **Send-receive data** (dark blue): processes send spiking data to each other. **Computation** is noted in red. The cost of the send-receive data is determined by the number of neurons on two communicating processes (N_a and N_b), how densely connected those populations are ($d\_conn_{a\_b}$ and $d\_conn_{b\_a}$) and the frequency at which those populations communicate (F_{a_b} and F_{b_a})

3. Data exchange (actual send and receive of spiking data). The cost of the send-receive data is determined by the number of neurons hosted on two communicating processes, their interconnection density and the frequency of communication, i.e. the firing rate.

3.2.2 Bottleneck to scalability

Past approaches to parallel and distributed neuronal simulators have focused their efforts on load balancing [86, 137], and not on the communication overhead of increased parallelism. However, in large scale SNN distributed simulations, propagation of spikes between processes has been suggested as the bottleneck for scalability [36, 229]. To demonstrate this, figure 3.2 shows a strong scaling experiment[1] of the Cortical Microcircuit [177] distributed simulation with frequent communication (at every time step) and random allocation of workload, i.e. neurons to processes. Whilst computation shows near-perfect scaling (Figure 3.2B), the simulation time scales poorly after a point (figure 3.2A). Looking at the proportion of time the simulation spends in each phase, figure 3.3A shows how the communication (both implicit synchronisation and data exchange) quickly becomes the dominant part and acts as a bottleneck to scalability.

Jordan et al. [124] and Schenck et al. [192] suggest that spike propagation may not be limiting scalability for simulations that make effective use of message packing. Event-message packing is one of the spike delivery optimisations [160, 86] that was discussed in section 2.4.1.2, in which simulations share spikes only at certain discrete intervals and not when they occur, reducing the burden of communication overhead. Since such simulations employ synaptic delays that are larger than the

[1]The details of the 77k neurons model and SNN simulator employed are described in section 3.3 and section 3.4.1.

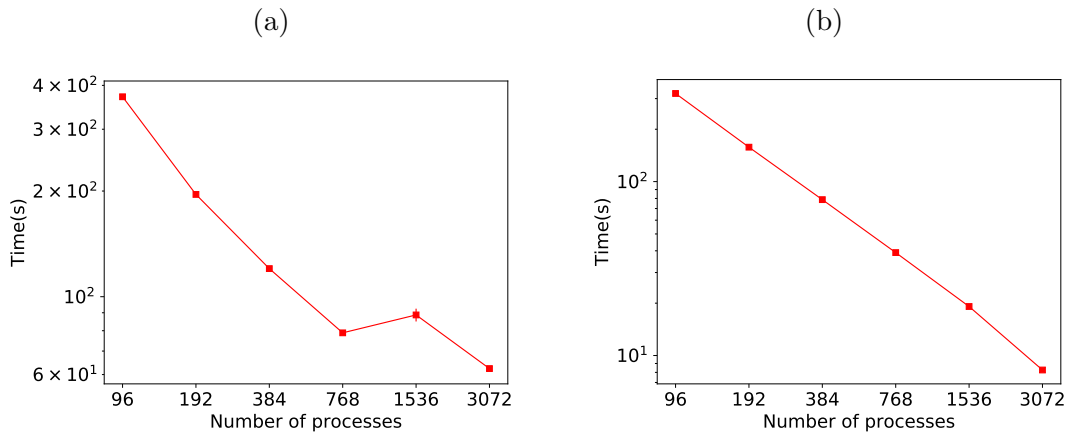


Figure 3.2: Evidence of the bottleneck to large-scale distributed SNN simulations scaling. Graphs show strong scaling results for a simulation with the same communication and computation time step, δ_{syn} (synaptic delay) and τ_{step} (time step) both set to the same value (0.1ms), which is known to be the worse case scenario in communication. **A**: Simulation time as the number of processes is increased. **B**: The time the simulation spends in computing.

resolution of the simulation time step, communication steps only occur on multiples of the global minimum synaptic delay. This significantly reduces the number of synchronisation events and therefore minimises the impact of communication on the overall simulation time. However, event-message packing does not solve the issue but rather pushes it to the right —i.e. it manifests at larger distributed scales. Figure 3.3B shows the same Cortical Microcircuit simulation but with event-message packing. Synaptic delay is set to 0.8s, with a timestep of 0.1ms, resulting in messages being sent every 8 time steps. The proportion of time spent in useful computation (red) still decreases as the parallelisation is scaled up, albeit at a lower rate.

Therefore, spike propagation is an important limiting factor for distributed SNN simulations, more so for models that cannot make use of message packing (such as the Cortical Microcircuit (CM) and Macaque Visual Cortex (MVC) models with random synaptic delays).

We define **computational efficiency** as the proportion of time a distributed simulation spends in the computation phase. Figure 3.2A shows overall simulation time stagnation in strong scaling experiments, as a result of poor communication scalability. Even though the computation part of the simulation scales well (Figure 3.2B), the benefits of increased parallelism are limited by communication —as shown by the shift in time spent in communication and computation.

3.3 Custom SNN simulator testing framework

The goal of this research project is not to develop a novel simulator, but to explore new algorithms and methods to optimally divide workload in neuronal simulations as well as to identify appropriate communication patterns. Current simulators do not permit custom allocation of work (see section 2.4.2.1) or alternative communication patterns, thus we have developed our own framework to measure the appropriateness of the strategies proposed. This constitutes contribution **C4** of the thesis.

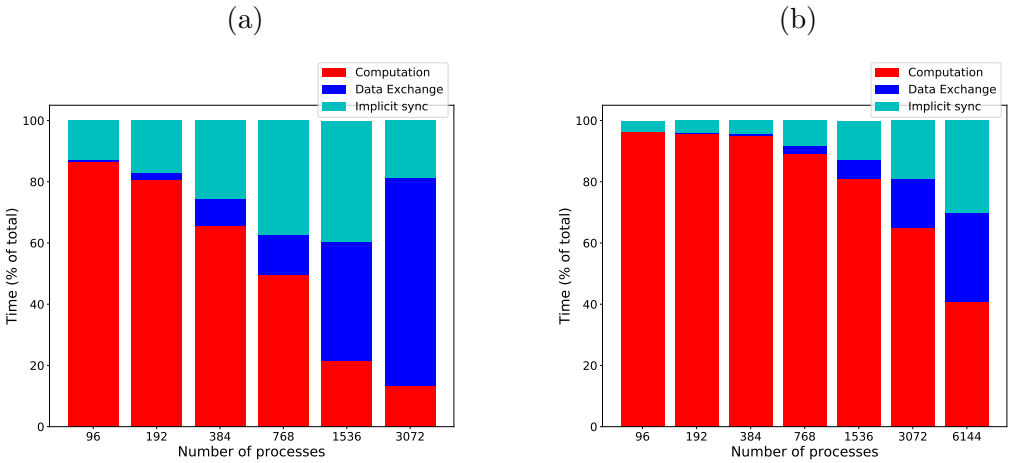


Figure 3.3: Evidence of the communication problem in large-scale distributed SNN simulations by showing the proportion of time SNN simulations spend in each phase (computation, synchronisation and communication). **A**: SNN simulation with frequent communication (no event-message packing). **B**: SNN simulation with event-message packing (delivery every 8 time steps). Both simulations show how communication (shades of blue) scales poorly and becomes the effective bottleneck to scaling. Using event-message packing pushes the issue to the right (larger distribution scales) but does not solve it.

At present, the computing performance of the simulator framework is not the primary concern and it does not attempt to compete with other high performance simulators in raw simulation time. The framework is be used as a test-bed for load balancing and scheduling algorithms to provide correct comparative results in both communication weight and computational balance. The findings from this research could and should inform the development of novel workload distribution strategies on other simulators.

3.3.1 Main components

The simulator framework is written in C++ and it has three stages:

1. Neuron initialisation and network construction.
2. Distributed sequential simulation in discrete time steps.
3. Gathering of statistics.

During phase 1, users write the models to be simulated. Three elements must be defined: populations (groups of neurons of a single type), connections (links between populations) and injections (regular input to populations during simulations). Phase 2 executes the simulation in distributed incremental time steps, whereas phase 3 gathers network activity to produce statistics regarding the simulation (such as spike rates, inter-spike intervals, simulation time and neuron and compute node communication).

The framework follows the general process of a discrete-time neuronal simulator consisting of three phases that are repeated at each time step: **update agents**,

update connections and **propagate messages**. In the case of a neuronal network simulation, *update agents* solves the partial differential equations that model the membrane potential of neurons, *update connections* solves the equations that model synapses and *propagate messages* transports spikes from pre-synaptic to post-synaptic neurons.

At its core, the framework is built to support distributed computation via message passing parallelism with MPI. Although the current version targets CPUs exclusively, the framework could be extended to support other computational paradigms (such as GPUs).

In order to allow testing of various communication and workload distribution strategies, the framework follows a modular design via a series of interfaces to specify the behaviour of the simulator:

- *Communication* interface: How the propagation of messages between MPI processes is performed.
- *Mapping of agents and connections to processes* interface: How the workload is distributed amongst MPI processes.

In addition to those interfaces, the framework exposes two more to define the agent model (neuron) and communication link model (synapse), which facilitates the use of the framework for other domains outside neuroscience.

To facilitate benchmarking, the framework keeps probes to measure the computation and communication in different parts of the simulator via MPI profiling API.

The code base for the simulator framework is open-source and can be found at `https://github.com/cfmusoles/distributed_sim`.

3.3.2 Validation

To validate the behaviour of the simulator framework, the SNN model presented by Vogels and Abbot [218] is implemented and executed. The VA model is chosen due to its simple formulation and ability to be easily scaled. It is a widely used benchmark in computational neuroscience and hence the expected behaviour is well defined.

The model, shown in figure 3.4, consists of a group of Leaky-Integrate and fire (LIF) neurons split into two populations, excitatory and inhibitory, based on the nature of their synapses. The model can accommodate any number of neurons as long as the excitatory to inhibitory ratio is kept to 4 to 1. Both populations are connected to themselves and to each other with fixed probability (0.02).

In the validation tests the number of neurons is set to 4000, with 3200 excitatory and 800 inhibitory. The simulation is run for 1 second with a 0.1ms time step. Neuron and synaptic parameters are left as those originally proposed by [218].

To validate the framework, a *PyNN* implementation (using *NEST* as back-end) [2] of the VA model is executed with the same parameters to produce a series of metrics commonly used to describe neuronal network behaviour at the system level:

[2] from `https://github.com/NeuralEnsemble/PyNN/blob/master/examples/VAbenchmarks.py`

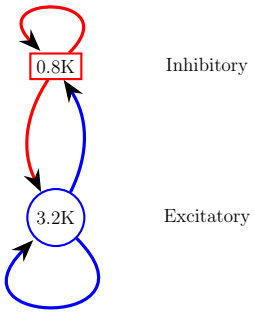


Figure 3.4: Diagram of the Vogels and Abbott model. Blue for excitatory connections, red for inhibitory connections; circles for excitatory populations and boxes for inhibitory populations.

- *Inter-spike interval* (ISI): histogram displaying the measured time between spikes across neurons in all populations.
- *Coefficient of Variation of the ISI* (CV-ISI): measurement of the distribution of the ISI (calculated as the standard deviation over the mean of the ISI per neuron).
- *Spike frequency* per population: Average spike rate per second per population.

Figure 3.5 shows qualitative similarities between the neuronal activity recorded by both simulators in one run. Both excitatory and inhibitory populations exhibit similar patterns of activity, with comparable Inter-spike interval counts and distributions (as shown by the CV-ISI graphs). Average spike frequency recorded across 5 runs (with random network connectivity but maintaining the parameters) on both simulators is shown in table 3.1 and indicate similar network activity in both populations.

Due to the internal random nature of the VA model (with random neuron connectivity and probabilistic initialisation of neuron membrane voltage), the behaviour expressed by the network is not expected to be identical on each simulation, but the general population trend must be kept. These results validate the behaviour of the simulator framework and are consistent when run distributed across multiple processes.

Table 3.1: Average spike rate in both simulators

	PyNN	***Simulator framework***
Excitatory rate	5.486 ± 0.222 Hz	5.626 ± 0.212 Hz
Inhibitory rate	5.579 ± 0.032 Hz	5.514 ± 0.041 Hz

3.4 Improving communication in distributed SNN simulations

This section presents work that targets the three phases of communication to improve overall computational efficiency in distributed simulations: implicit synchro-

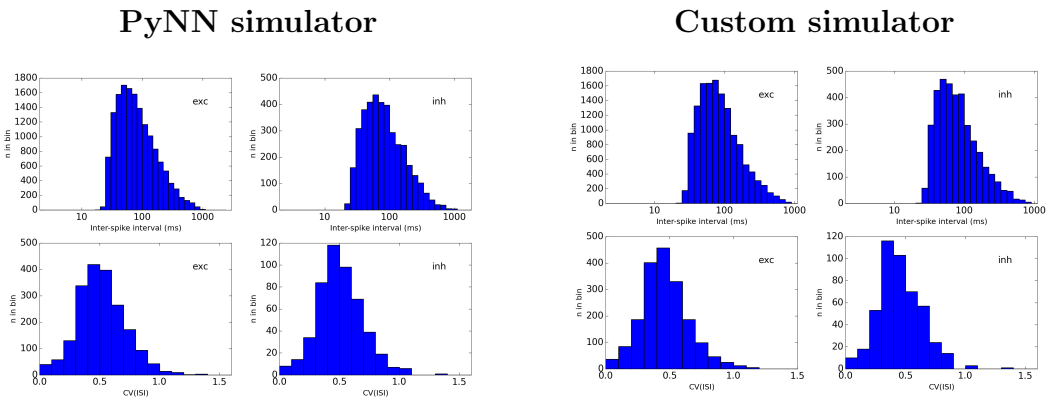


Figure 3.5: Inter-spike intervals (ISI) and Coefficient of Variance of ISI (CV ISI) counts from 1 second simulation of the VA model. *PyNN* implementation (top) and our simulator (bottom).

nisation, process handshake and data exchange. We introduce a connectivity-aware allocation of neurons to compute nodes by modelling the SNN as a *hypergraph*. Partitioning the hypergraph to reduce interprocess communication increases the sparsity of the communication graph (contribution **C2**). We propose dynamic sparse exchange as an improvement over simple point-to-point exchange on sparse communications (contribution **C3**). Results show a combined gain when using hypergraph-based allocation and dynamic sparse communication (contribution **C1**), increasing computational efficiency by up to 40.8 percentage points and reducing simulation time by up to 73%. The findings are applicable to other distributed complex system simulations in which communication is modelled as a graph network.

In other complex system simulations (such as molecular interactions, fluid dynamics, etc.), computation is divided based on position and communication tends to happen primarily (or exclusively) in the overlapping or adjacent locations. Communication in parallel SNN simulations is very different since neurons can communicate (i.e. spike) with potentially any other neuron in the model, irrespective of their location. The format of the communication is also unique, as it takes the form of low frequency, discrete messages (spikes) from one neuron to all of its post-synaptic targets. These two facts make communication in SNNs very **dynamic** (target neighbours change across the simulation) and **sparse** (target neighbours at each communication step are a subset of the total). Hoefler et al. [111] proposes *Dynamic Sparse Data Exchange* (*DSDE*) algorithms for scalable sparse communication in large neighbourhoods with good scalability. This section demonstrates the benefits of *DSDE* as a communication pattern in SNN simulations.

The allocation of neurons and communication strategies are evaluated using a scientifically relevant model: the Cortical Microcircuit. This models is well suited to evaluate allocation and communication strategies in SNN simulations for the following reasons: it is arbitrarily scalable; its scalability is communication-bound when using random synaptic delays (as shown in figure 3.3); and it is a very popular model in neuroscience [199, 177] and its structure is similar to other more complex models ([195, 119, 9, 10]).

3.4.1 Cortical Microcircuit SNN model

For the benchmark experiments, the model used is based on the Cortical Microcircuit (*CM*) described by Potjans and Diesmann [177], scaled to 77k neurons and 150M synaptic connections. The *CM* model is representative of microcircuit-type models in which a SNN is constructed by layers of neurons and connectivity probabilities within and between layers. As such, findings on the CM model can be applicable to other microcircuit models and similarly structured biologically plausible networks.

The size of the model is sufficient to display the limitations of scalability in distributed simulations due to communication overhead —see Figure 3.3. This is a consequence of the high frequency of synchronisation required in the simulation: event-message packing optimisation is not allowed because the model contains random synaptic delays, forcing processes to synchronise at each time step.

To ensure network activity across the simulation with resting potential (-45mV) and spiking threshold (-50mV). The choice of having a higher resting potential than a spiking threshold is justified in [37] for current-based connections —note that the resting potential (natural value the neurons tend towards in absence of input) is not the same as the reset potential (potential value set after spiking). All neurons receive a constant current of 0.95mA, which differs from [177] Poisson spike trains but results in an equivalent global average activity of 5–7 spikes/s. A constant input is more advantageous in our scenario since it is simpler to implement than a Poisson spike train; previous work has demonstrated that an equivalent activity pattern can be achieved with either approach [216].

All experiments shown in this chapter involving increasing number of processes are strong scaling experiments in which the size of the CM model is constant as the number of processes increases.

3.4.2 Hardware architecture

Simulations of the CM model are run on the ARCHER Cray XC30 MPP supercomputer, with 4920 nodes, each with two 12 cores Intel Ivy bridge series processors (for a total of 24 cores) and up to 128 GB RAM available. Computing nodes are connected via fast low latency Cray Aries interconnect links [3].

ARCHER allocates exclusive computing nodes (cores and memory), however, as a cluster computer, network related resources are potentially shared. There are two types of noise that can affect benchmarking results: external application traffic contention and distance on the allocated computing nodes. Traffic noise is minimised by running each iteration (same seed) of an experiment multiple times (with the same node allocation) and selecting the fastest sample. Node distance variability (communication between any two nodes depends on how close they are in the topology) is smoothed by running each set of experiments multiple times with different node allocations. Furthermore, when comparing communication and allocation strategies, all candidates within an experiment set run with the same node allocation.

[3] More information regarding ARCHER hardware can be found on `https://www.archer.ac.uk/about-archer/hardware/`

3.4.3 Hypergraph representation of SNNs

Communication in distributed SNN simulations happens due to pre- and post-synaptic neurons being hosted by different processes. Figure 3.6 shows the Process Communication Graph (PCG) resulting from a mapping of neurons to 8 processes. In the PCG, edges are shown only if there are *any* synapses between neurons of both populations. As a consequence, during simulation, each process has to synchronise (i.e. receive information) with every neighbouring process it has in the PCG.

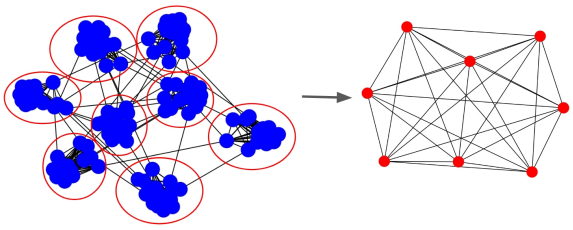


Figure 3.6: Process Communication Graph (PCG) that represents parallel SNN simulation communication. Left: the graph with blue nodes represents the SNN synaptic connectivity and the red circles are processes to which the neurons are mapped. Right: Resulting process graph in which edges represent processes that need to synchronise during simulation, i.e. have an inter-process synaptic connection between themselves. The PCG describes the process neighbourhood for each computing node.

This work tackles the communication overhead issues that limit distributed scalability in large scale SNN simulations. The goal is to increase overall computational efficiency by reducing the time simulations spend on the three phases of communication between processes. We focus on reducing the overhead of communication by: 1) using a connectivity-aware allocation of neurons to compute nodes; and 2) employing scalable sparse parallel communication patterns. These two strategies are complementary and address the sparsity of communication in the PCG.

Since communication in distributed SNN simulations only happens when there is inter-process connectivity in the PCG, minimising this connectivity directly reduces the communication requirements. The topology of biological plausible complex neuronal networks is found to show presence of clusters, where local connectivity is expected to outnumber remote connectivity [39, 170]. Hence, there is an opportunity to optimise communication by considering this clustering when assigning neurons to compute nodes.

A hypergraph is proposed as a model of the SNN to increase the sparsity of the PCG. A hypergraph has been shown to successfully model total communication in parallel applications [63, 61]).

A SNN can be thought of as graphs with neurons as nodes and synaptic connections as edges. To better model the cost of communication, instead of using edges (one to one connectivity) we use hyperedges. Each hyperedge contains a pre-synaptic neuron and all of its post-synaptic targets —see Figure 3.7A. This captures the all-or-nothing communication behaviour of neurons, where when a neuron spikes, a message is sent to all its post-synaptic targets (and not a subset of targets). Thus, a hyperedge adequately models the unit of communication in a SNN. To model workload, hypergraph nodes are weighed by the number of dendritic inputs,

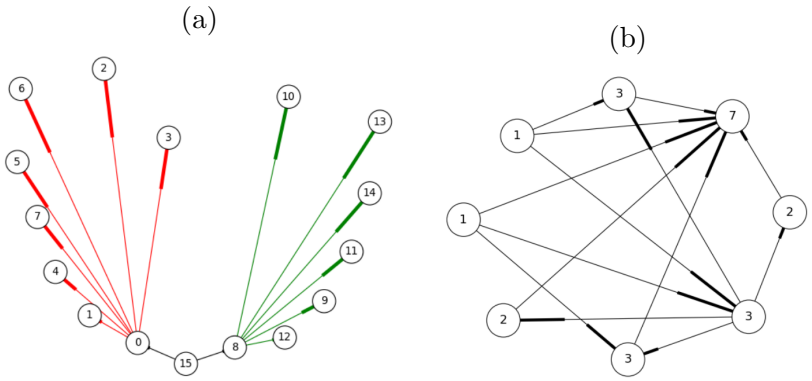


Figure 3.7: Representing the SNN as a hypergraph. **A**: three hyperedges (black, red and green) shown, where a hyperedge includes a pre-synaptic neuron and all its post-synaptic neurons (the numbers describe neuron IDs). This representation corresponds to the manner neurons communicate, i.e. when a neuron fires all its post-synaptic targets need to be notified. **B**: The hypergraph nodes are weighed by the number of dendritic inputs to a neuron as an estimate of the workload located at each process. In this example, edges represent synaptic connections between neurons (nodes), with the thicker end indicating direction. The number on the node represents the weight associated to each neuron, equal to the sum of its input plus one.

following a similar approach proposed by [106] for workload balance —Figure 3.7B. The number of inputs is a good estimate of the workload associated with hosting a neuron; synaptic objects handling dendritic input computation are located on the post-synaptic side, therefore more dendritic input requires more computation during the synaptic update phase.

3.4.4 Hypergraph partitioning for neuron allocation

When neurons within the same hyperedge are assigned to different nodes of the PCG, communication between them is required during simulation. Note that inter-process communication is therefore not required if there are no hyperedges spanning more than one process. Hence, the allocation of neurons to processes can be formulated as an optimisation problem where the goal is to reduce the number of hyperedges cut between processes. Multilevel hypergraph partitioning, generally understood to produce better partitions faster than other alternatives [103], is used. See section 2.7.2 for an in-depth review on hypergraph partitioning algorithms. This work makes use of the state-of-the-art partitioning library *Zoltan* [31], with the agglomerative inner product matching coarsening method [47] and FM refinement method [76]. To better represent the communication costs, each hyperedge cut is weighed based on the number of participant parts minus one. Formally, the total cost of a partition scheme is $\sum_{i=0}^{|E|} P(e_i)$, where $e_i = \{p_1, p_2..., p_n\}$ represents the set of partitions that contain any node in hyperedge e_i, and E is the set of all hyperedges. The cost $P(e)$ of hyperedge e is defined as $P(e) = |e| - 1$.

To avoid trivial solutions that minimise the hyperedge cut (such as assigning all vertices to one partition) partitioning algorithms maintain load balancing by only allowing solutions that have a total imbalance factor that is below a specified

tolerance value (in our experiments, 1.001). The workload of each node (neuron) in the hypergraph is estimated by the number of incoming post-synaptic connections, and using that as the weight of the node (see Figure 3.7). The computation in the simulation is dominated by synaptic updates, based on experimental validation. Post-synaptic objects live in the same partition local to the post-synaptic neuron, hence those partitions with more incoming synapses will have more computation to perform during updates.

3.4.5 Results of the hypergraph partitioning-based allocation

The baseline neuron allocation strategy *Random Balanced* is a variation of random allocation that takes the number of post-synaptic connections into account to keep processes balanced. *Random Balance* performance is compared to *Hypergraph*, our allocation strategy that uses hypergraph partitioning to minimise the total interconnectivity between nodes in the PCG. Both strategies use standard point-to-point *PEX* to communicate spiking data. As reviewed in section 2.3.2 and 2.4.1.2 this is a common pattern used for P2P communication in distributed complex system simulations.

Figure 3.8 shows comparative results from both allocation strategies. Since they are based on the same metrics for load balance, there are no differences in implicit synchronisation time between strategies (Figure 3.8A). *Hypergraph* has its impact in terms of interprocess connectivity. As expected, the number of average runtime neighbours (ARN, average number of target processes at each communication step per process) is reduced (Figure 3.8C), making **communication more sparse**. This leads to fewer remote spikes, when a local neuron spike needs to be propagated to other processes (Figure 3.8D) and a decreased total amount of data exchanged (Figure 3.8E). Despite the improvement in sparsity, data exchange time (Figure 3.8B) is not significantly reduced and therefore simulation time is not affected since *PEX* is not designed to take advantage of sparsity —see discussion in section 3.4.10.1.

3.4.6 Dynamic sparse data exchange communication to improve distributed simulation scalability

As discussed in section 2.3.2, communication in SNN simulations falls into the category of *census*, a common parallel programming function in which a process receives a piece of data from each of all other processes. Personalised census or personalised exchange (*PEX*), the most basic implementation of census [111], occurs in two steps: inter-process handshake and send and receive data. During handshake, processes inform their targets that they will be sending data to them. In the second phase, each process post data and listens to messages only from those processes. Those phases are depicted in figure 3.1.

With respect to scalability, there are two issues with *PEX*: on the one hand, due to the dependency between phases, they occur sequentially, i.e. each process must wait until it has completed the handshake before sending data. This adds waiting time that is dependent on the total number of processes (messages will take longer to propagate in larger topologies). On the other hand, during handshake, each process sends metadata to all others, even if they do not need to send spiking data to them

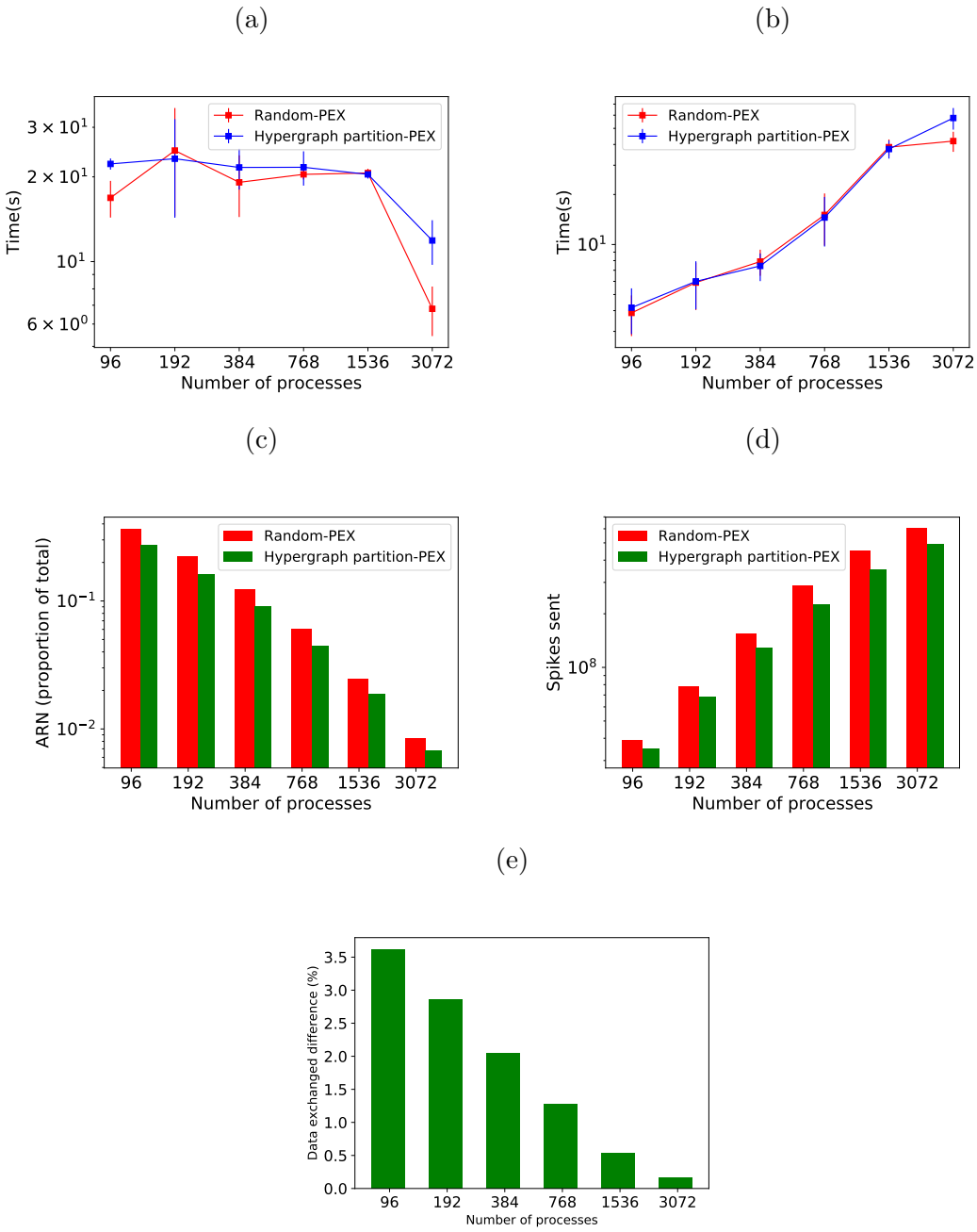


Figure 3.8: Performance results of *Random balanced* allocation compared with *hypergraph partitioning*. The top part shows quantitative timings during the different phases of communication: **A** implicit synchronisation time and **B** data exchange time. The bottom part of the figure shows reductions brought by hypergraph partitioning over random: **C** Average number of Runtime Neighbours (ARN, average number of processes each process communicates to at any given communication step, a subset of the total neighbourhood defined in the PCG); **D** remote spikes (local spikes that need to be propagated to other processes); **E** spiking data volume exchanged difference.

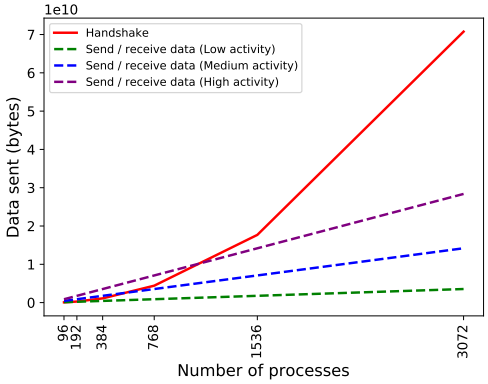


Figure 3.9: Theoretical data volume exchanged in the two phases of communication by *PEX* on an artificial network with constant spiking activity (20 spikes/s for low activity, 80 spikes/s for medium activity and 160 spikes/s for high activity). During spiking data exchange, processes send data to each other in two phases: handshake (coordinating intention to send information) in solid red line; and send-receive spiking data, in dashed lines. The send-receive exchange volume is dependent on the activity density (spiking average) per process. The communication profile of the simulation and the number of processes used determines which phase is dominant.

(they still need to inform others that they will not be receiving data). This causes an overhead of metadata with a quadratic growth with the number of processes, shown in figure 3.9, and quickly becomes the dominant part of the exchange.

Neighbourhood Exchange (*NBX*) is a type of *Dynamic Sparse Data Exchange* algorithms proposed by Hoefler et al. [111]. It targets the overhead caused by the handshake phase of communication by overlapping it with the send-receive data phase. For sparse communication, Hoefler et al. [111] demonstrate that *NBX* performs better than *PEX*.

Previous attempts to optimise data exchange in SNN simulators have been limited to implement *PEX*-like point-to-point communication [135, 98, 159, 124, 9]. To the best of our knowledge, no SNN simulation has considered using *DSDE* algorithms. It is proposed here that *NBX* alongside hypergraph partitioning can exploit the increased sparsity of communication of the PCG to reduce the overheads of communication in all its phases.

3.4.7 Implementation of *PEX* and *NBX*

To commpare *PEX* and *NBX* communication patterns, both algorithms are implemented as communication methods on the custom SNN simulator framework.

Both algorithms require each process to maintain a lookup table of target processes each local neuron connects to (populated before starting the simulation). When a process is in the communication stage, it matches the local spiking neurons to the lookup table to generate the data to be sent to each process. PEX and NBX differ in the way they distribute this data. Both *PEX* and *NBX* follow the implementations described by Hoefler et al. [111].

For the comparisons, *PEX* is implemented with an all to all call for the hand-

shake, which informs other processes of which processes to listen to. An asymmetric all to all follows, where each process may send different amounts of data, to send-receive data (which is preferred to individual send and receive postings per process as it incurs in less call overheads).

NBX is implemented as depicted in Figure 2.2. First, each process sends data asynchronously (non-blocking) to all its targets. Whilst the messages are being delivered, processes probe for incoming messages. Once a sender process is notified that its messages have been received, it places an asynchronous barrier and continues to probe for incoming messages until all other processes have reached the barrier (with an MPI_Ibarrier call), signalling the end of the communication phase. The use of an asynchronous barrier removes the need for an explicit handshake, effectively allowing the data exchange to start without explicit knowledge from the receiver.

3.4.8 Results of NBX as communication pattern for SNN simulations

NBX and *PEX* communication patterns are compared in simulations using *Random Balanced* allocation. Figure 3.10 shows how *NBX* reduces communication time by significantly decreasing implicit synchronisation time (Figure 3.10A), as a result of a more balanced communication time amongst processes —see discussion in section 3.4.10.2. Total amount of spiking data sent across processes is decreased (Figure 3.10B) due to the elimination of the handshake phase during communication. Data exchange time is not impacted (Figure 3.10C) despite the reduction in data —see discussion in section 3.4.10.2.

3.4.9 Using hypergraph partitioning to increase the effectiveness of *NBX*

Once the hypergraph partitioning allocation and the dynamic sparse data exchange strategies have been evaluated independently, we combine them to identify synergies. The four candidates compared are *Random-PEX*, *Random-NBX*, *Hypergraph partition-PEX* and *Hypergraph partition-NBX*, indicating which neuron allocation algorithm (random balanced or hypergraph partitioning) and communication strategy (*PEX* or *NBX*) is used.

The results in Figure 3.11 show the effectiveness of *Hypergraph partition-NBX* over the rest, with reduced simulation time (Figure 3.11A) as a consequence of decreased communication time. The improvement is due to implicit synchronisation time gain (Figure 3.11B) brought by the use of *NBX*, as well as a reduction of data exchange time of *Hypergraph partition-NBX* over *Rand-NBX* —see discussion in section 3.4.10.3 for an in depth analysis.

The *Hypergraph partition-NBX* approach is not only faster than the random alternatives, but as figure 3.11A demonstrates, it also manages to improve scaling by delaying the point at which the increased communication overhead outweighs the gains in simulation time. The approach scales up to 1536 processes for this model (per 768 of the other alternatives).

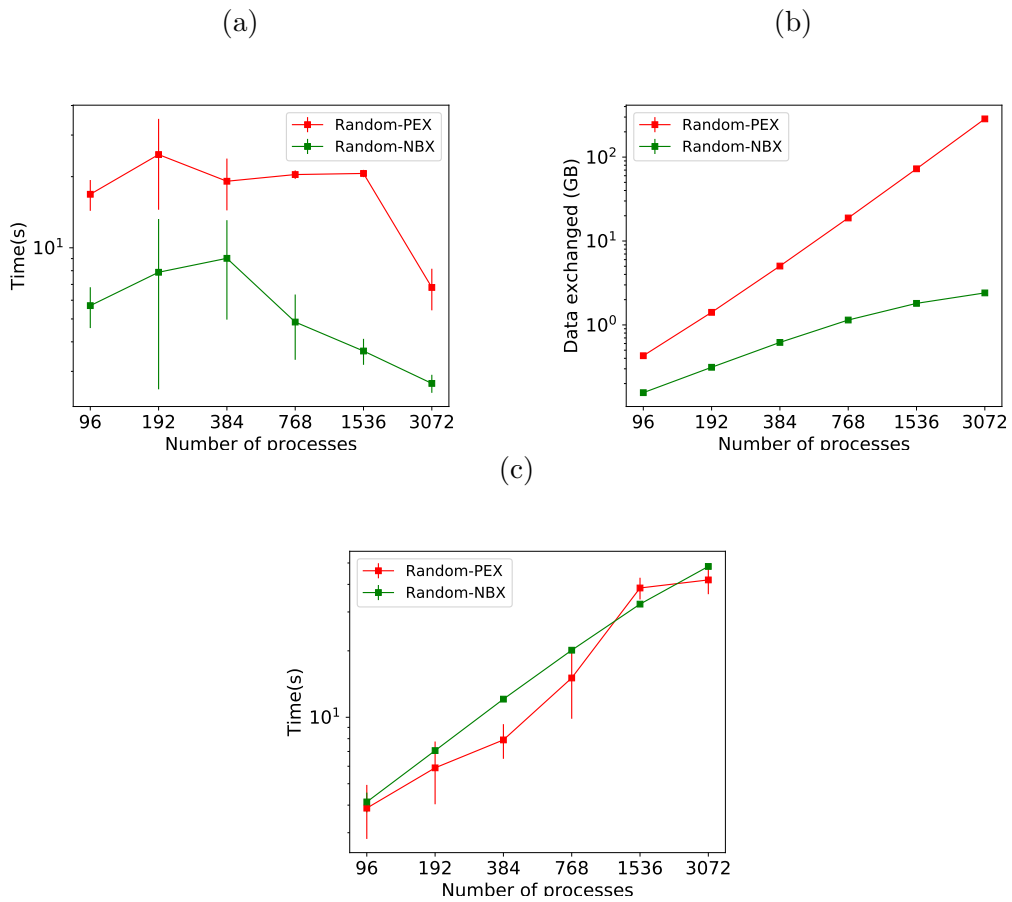


Figure 3.10: Communication time using *PEX* and *NBX* are shown. **A**: implicit synchronisation time per simulation. **B**: total volume of spiking data exchanged during simulation. **C**: data exchange time per simulation.

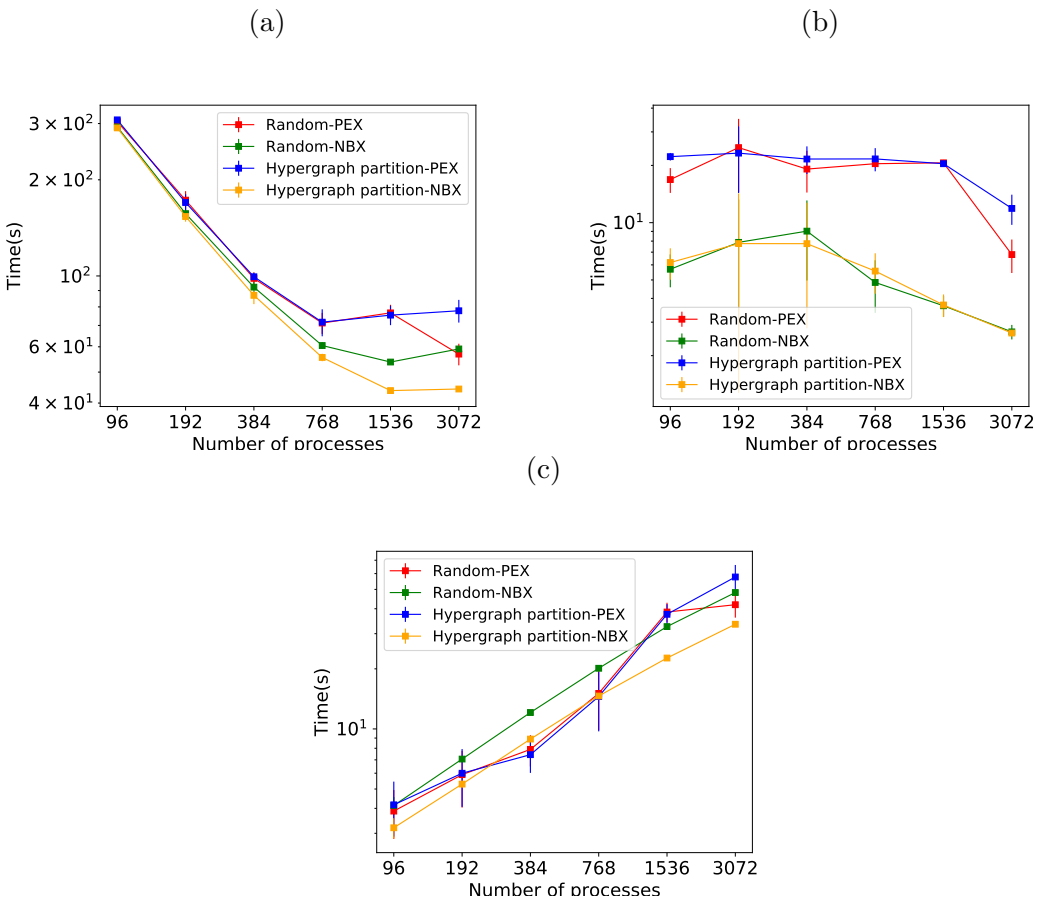


Figure 3.11: Simulation performance measurements for *Rand-PEX*, *HP-PEX*, *Rand-NBX* and *HP-NBX*. **A**: overall simulation time. **B**: implicit synchronisation time per simulation. **C**: data exchange time per simulation.

3.4.10 Discussion of hypergraph partitioning and NBX results

Most SNN simulators allow users to describe their models at population level. Biological data at that level of detail is common [195, 218, 177, 8] and therefore it is natural to use when building simulations. Regardless of the model definition, at some point the simulation needs to build representations of neurons and connectivity, at which level we can now apply our partitioning approach. Alternatively, because vertices in the hypergraph are an abstraction, they can be used to represent groups of neurons such as populations and regions. Partitioning can be performed at any level of detail, but doing it at the neuron level allows finer control of workload balance by assigning a few neurons to nodes that are almost full to capacity, instead of having to assign whole populations. It also allows better communication optimisation as the algorithm treats individual connections instead of aggregations that would be necessary when using higher levels of abstraction. Moreover hypergraph partitioning can find optimal intra-population divisions that may not be apparent to the user in models at the brain-scale, should they have to manually allocate workload to computing nodes (both impractical in larger models and suboptimal).

3.4.10.1 Hypergraph partitioning as a neuron allocation strategy

Our proposed neuron allocation method based on hypergraph partitioning offers an alternative to approaches employing graph partitioning [102] or clustering [214]. A hypergraph allows the total communication volume of the simulation to be modelled more accurately than using a normal graph [63, 61]) which enables a better allocation of neurons to computing nodes to reduce connectivity between processes.

Hypergraph partitioning is effective in minimising interprocess connectivity, as shown by a 10–20% reduction in remote spikes (Figure 3.8D) and 20–25% in average runtime neighbours (Figure 3.8C). Increased sparsity of the communication graph, however, does not impact data exchange time —see figure 3.8B. This is because *PEX* is not designed to take advantage of this sparsity: send-receive data is implemented with MPI_Alltoall, which results in global synchronisation of all processes even with reduced interconnectivity. Therefore, performance improvement on simulation time is negligible.

Partitioning alone brings a very moderate reduction in volume of data exchanged (Figure 3.8E) that becomes negligible as the parallelism is increased. The effect is expected since partitioning optimises only the data send-receive portion of the exchange phase, and as shown in Figure 3.9 this is not the dominant part of the communication, less so as the number of processes increases. Parallel SNN simulations in which the send-receive part is dominant (e.g. high frequency of neuronal activity or increased neuronal density per process) could see a significant improvement here.

Lytton et al. [147], point out that simple collective communication patterns like *all gather* may perform better than point-to-point provided the number of ranks is significantly lower than the average number of connections per neuron. This tipping point assumes random allocation of neurons and hence random chance of post-synaptic targets being placed at any computing node. Using partitioning actively increases the probability of a pre- and post-synaptic neuron living in the same process (as seen by a reduction in runtime neighbours in Figure 3.8C) and therefore neurons will normally have less targeted process than post-synaptic targets. This

shifts the tipping point in favour of point-to-point communications to be reached much sooner, at fewer processes.

3.4.10.2 *NBX* dynamic sparse exchange

NBX has a strong impact on implicit synchronisation time (Figure 2.2A), which is an indirect measurement of load balance between processes. With both alternatives employing *Random Balanced* allocation, the computation phase remains the same. Therefore any change in load balance, reflected in implicit synchronisation, can be attributed to the communication strategy —some processes taking longer than others to finish communication phase which carries over to the next communication step when processes re-synchronise again. *PEX* makes use of collective MPI calls, which ensure that processes are synchronised at the point of entry of that function. Note that each process may continue execution as soon as it has receive messages from all others. This is not guaranteed to happen at the same time for all processes, introducing imbalance —a well known phenomena in parallel synchronisation [110]. With *NBX* processes are implicitly synchronised with an asynchronous MPI_Ibarrier: all processes are guaranteed to have sent all data before any of the processes finish the communication phase and continue with the simulation. This acts as a balancing mechanism whilst not forcing processes to be idle, since the barrier notification messaging happens in parallel whilst receiving data.

The cost of synchronisation for *NBX* shows in the measured time for data exchange on Figure 2.2C. *NBX* scales in a more predictable way than *PEX*, but at the level of communication of the simulation (random allocation of neurons leads to almost all to all process connectivity) it is often slower. Hoefler et al. [111] indicates that the performance of *NBX* is dependent on the number of neighbours (sparsity) during communication. Hence, increasing sparsity would improve *NBX* performance —see discussion in section 3.4.10.3.

Figure 2.2B shows volume of data sent by both alternatives, with a qualitatively improvement of *NBX*. This measurement accounts only for explicit data sent; for *PEX* this includes handshake metadata, which as we have discussed becomes dominant with increased parallelism (each process needs to send data to all other processes specifying whether further communication is to be expected). *NBX* performs the handshake implicitly with MPI_Ibarriers, which has a payload of 0 and does not require all to all messaging.

3.4.10.3 Synergy between Hypergraph partitioning and *NBX*

The combined *Hypergraph partition-NBX* strategy performs better than either one on its own (Figure 3.11A). It keeps the reduced implicit synchronisation time (Figure 3.11B) that comes with *NBX* better process balance during communication. Furthermore, when compared to *Random-NBX*, *Hypergraph partition-NBX* has enhanced communication sparsity due to partitioning allocation that increases the effectiveness of the sparse communication pattern, resulting in data exchange time reduction (3.11C). With sparsity, *Hypergraph partition-NBX* matches the data exchange performance of *PEX* alternatives in lower processor counts, and improves upon it as the parallelisation increases.

The dual nature of *Hypergraph partition-NBX* could impact performance in SNN models with different communication profiles. In high frequency communicating

SNN simulations (high neuron firing rate or high process neuronal density), where communication is dominated by data send-receive (Figure 3.9), the impact of hypergraph partitioning in reducing the volume of communication is expected to increase: when more neurons are spiking, more data is sent across processes, making any reduction in inter-process synapses more significant. This effectively increases the sparsity of communication, improving the effectiveness of *NBX*. In low frequency communicating SNN simulations (low neuron firing rate, low process neuronal density or a modular SNN model that can be partitioned well), where communication is sparse, *NBX* is expected to speed up data exchange time —as discussed in section 3.4.10.2 and by [111]. Therefore, with increased network communication sparsity, *NBX* is a more suitable communication pattern than point-to-point strategies used by SNN distributed simulators [159, 98, 135, 124].

3.5 Larger model with round robin allocation

We have shown that the proposed strategy leads to significant performance improvement on the *CM* model. The main limitations of this model are its relatively small size and the random connectivity between its layers, which can pose a challenge to workload distribution (particularly the random connectivity, as it makes reducing communication through partitioning more challenging). To demonstrate further applicability of the proposed strategy to larger and modular models, the effect of *HP-NBX* on the *MVC* model is evaluated. The *MVC* model has 660k neurons with 620 million synapses, arranged in 32 modular areas as described by Schmidt et al. [195]. The internal structure of each area resembles that of the Cortical Microcircuit model.

Round robin is a standard neuron allocation algorithm employed by many neuronal simulators (such as NEST or NEURON). Although it is an adequate load balancing approach [124], round robin represents the worst-case scenario in terms of connectivity, as it purposefully separates neurons which are more likely to be more interconnected (those belonging to the same population). As a consequence, round robin forces each process to be communicating with a high number of other processes.

The experiment in this section is performed on the same HPC system as described for the CM (ARCHER).

3.5.1 Macaque visual cortex multi-scale model

To understand how the hypergraph partitioning allocation and the *NBX* communication scale to larger models, we implement the multi-scale model of the macaque visual cortex (*MVC*) described by Schmidt et al. [195]. The model bridges the gap between local circuit models (such as the *CM*) and large-scale abstractions to represent one hemisphere of the macaque visual cortex. It is comprised of 32 areas, each one constructed as a *CM* and connected to others with fixed probability, i.e. a cell on one area is connected to cells in another area with a probability that is fixed for all cells between said two layers. The nature of the model is modular (i.e. divided in areas) which makes it a suitable candidate for *HP-NBX* allocation.

The model is implemented at 16% of the original scale, with 660k neurons and approximately 620 million synapses, with the same connectivity probability. Neu-

rons share the same parameters as the ones detailed in the original model [177]. As with the *CM* model, a constant current of .38mA is introduced to all neurons. To produce an average activity of 20–23 spikes/s, the weight multiplier of inhibitory connections is set to -15 (instead of -4).

3.5.2 Results of *HP-NBX* on the MVC model vs round robin

A set of strong scaling experiments has been performed to analyse how *HP-NBX* scales to larger, modular models, such as the *MVC* model. The experiments involve simulating the MVC model over a 700ms time interval with a 0.1ms timestep, using the following range of scales: 192, 384, 768, 1536, 3072 and 6144 processes.

Round robin scheduling of computations is a common workload distribution strategy in neuronal simulators. It is used as a simple load balancing approach, spreading neurons with similar activity patterns across the network [124]. Thus, round robin is used in here as a baseline, with *PEX* communication.

Figure 3.12 shows the results of the strong scaling experiments for the larger, modular MVC model. It compares our proposed strategy *HP-NBX* to the baseline *Round robin-PEX*. The computation time scales linearly with the number of processes (figure 3.12C), indicating the simulation does benefit from parallel execution. *HP-NBX* significantly impacts interprocess communication by reducing the number of average runtime neighbours by around 90% (figure 3.12B). Hosted neurons have to send spikes to fewer neighbours, and thus data exchanged is reduced in comparison (figure 3.12A). This is a qualitative reduction, as the data volume has a quadratic growth for *Round robin-PEX* but linearly for *HP-NBX*. With less data volume being sent and a more balanced communication pattern, both phases of communication are reduced: data exchange time (3.12D) and implicit synchronisation time 3.12E). As a consequence, *HP-NBX* simulation time scales well with the number of processes, whereas *Round robin-PEX* struggles with high process counts (no improvement with 1536 processes or higher).

The overall computational efficiency gain of *HP-NBX* (proportion of time spent in computation) is outlined in Figure 3.13, which compares it to the baseline *Round robin-PEX*. Not only overall simulation time is reduced by up to 73% (Figure 3.12F) but the proportion of time spent in computation is increased (Figures 3.13A and 3.13B), resulting in an **improved computational efficiency**, by up to 40.8 percentage points (from 22.2% to 63%).

3.6 Discussion of MVC results

3.6.1 *HP-NBX* vs *round robin* in large and modular model

HP-NBX is capable of taking advantage of the modularity of the model and decreases the average number of neighbours each process communicates with by 90% (figure 3.12B). The lower ARN impacts the data exchanged not only in reducing its volume but also allowing linear scaling with the number of processes, compared to the quadratic growth of *Round robin-PEX*.

The decreased ARN allows the *NBX* algorithm to be more efficient and balanced and hence simulations spend less time in implicit synchronisation (figure 3.12E). The

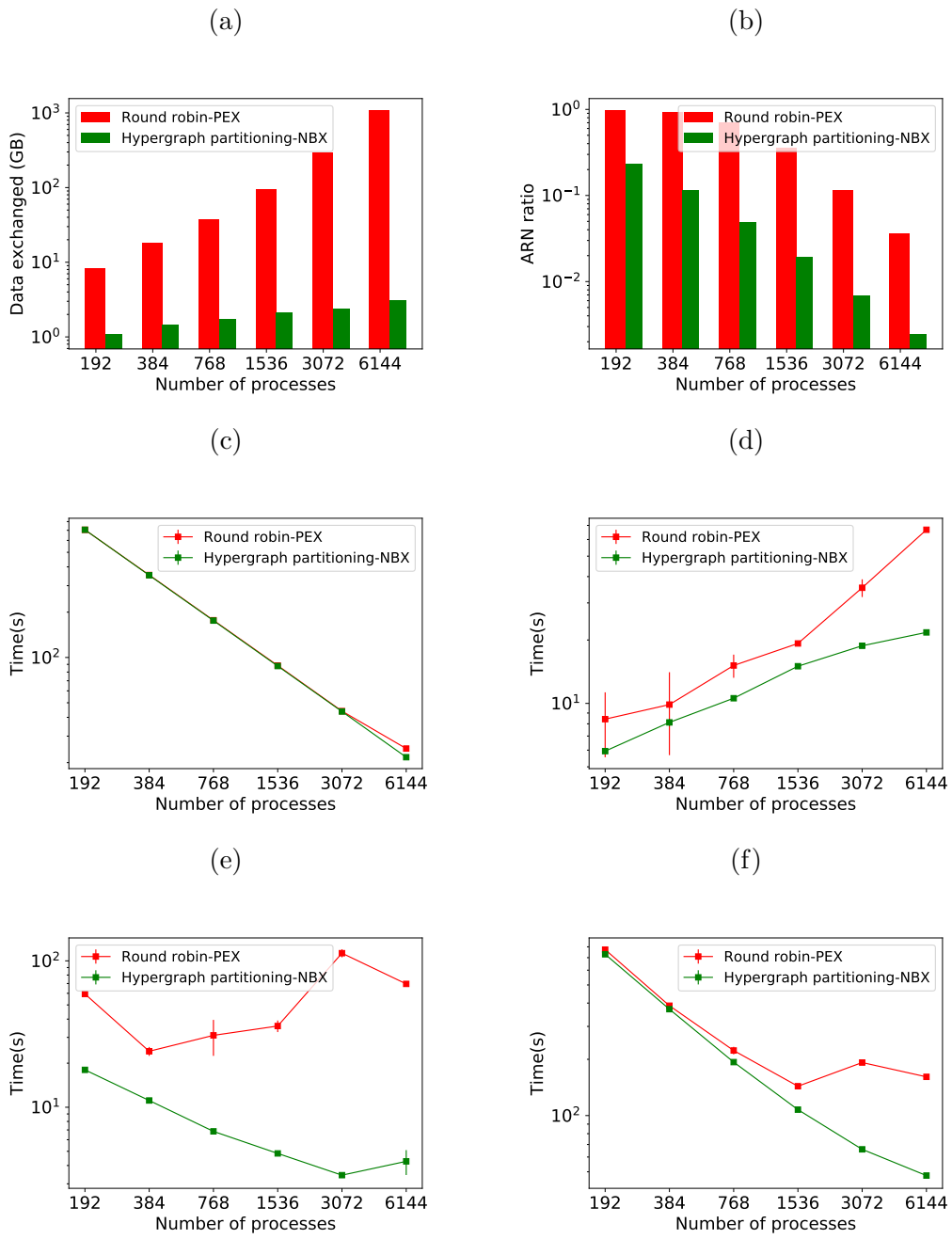


Figure 3.12: Simulation results of the *MVC* model showing performance improvements of *HB-NBX* over the baseline *round robin-PEX*. The top part of the figure displays connectivity-related metrics. **A**: data volume exchanged during simulation; **B**: average runtime neighbours in communication phases reduction in percentage. The bottom part of the figure shows results of the *MVC* scaling experiments with both alternatives. **C**: computation time which decreases linearly, demonstrating the potential benefit of increased parallelism; **D**: data exchange time; **E**: implicit synchronisation time; **F**: total simulation time.

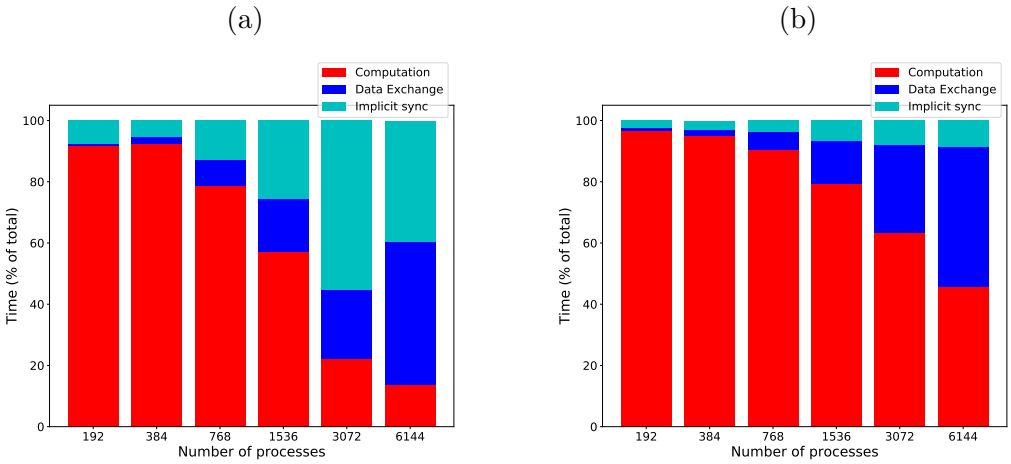


Figure 3.13: *HP-NBX* computational efficiency gains over the baseline*Round robin-PEX during MVC simulations*. The graph shows the proportion of time an MVC simulation spends in computation (red), implicit synchronisation (light blue) and data exchange (dark blue) when using *Round robin-PEX* (**A**) and *HP-NBX* (**B**). The proposed strategy *HP-NBX* consistently increases computational efficiency in all processes counts (percentage points improvement in increasing process count order: 5.2, 2.7, 12, 22.3, 40.8 and 32).

qualitative difference in scaling of data volume (linear for *HP-NBX* and quadratic for *Round robin-PEX*) leads to reduced data exchange time (figure 3.12D). Both figures 3.12D and 3.12E show how our approach scales better: data exchange time growth slows down with the number of processes, in contrast with a continuous quadratic growth of the baseline; similarly, implicit synchronisation time decreases with higher processor counts, whereas it is increased in the baseline. This limits the scalability of simulations with *Round robin-PEX* (figure 3.12F), with simulation time not being improved despite the extra computing resources (1536 processes and above). In contrast, *HP-NBX* scales well, with simulation time reduced for all process counts.

The impact of *HP-NBX* on data exchange time reduction contrasts with those seen when comparing random allocation and hypergraph partitioning using the *CM* model, in which a moderate reduction on average runtime neighbours (25%) or data volume (1–3%) on their own did not significantly impact data exchange time. There are two key factors that explain the difference. The first one is quantitative: the communication reduction is much greater in the *MVC* model (90% average runtime neighbours and 80% data volume). The second reason is qualitative and is due to the scaling of both data volume and ARN. The data volume difference (figure 3.8E) and the ARN difference (figure 3.8C) in the CM simulation decreases with the number of processes, whereas both differences increase rapidly in the *MVC* model, making it more significant the more the simulation is parallelised. This is therefore sufficient to impact communication time.

Overall figure 3.12 shows that *HP-NBX* is a more effective allocation and communication algorithm than *Round robin-PEX*, yielding shorter simulation times (figure 3.12F). It is worth noting that our implementation of round robin does not take advantage of short cuts such as finding the target process for a post-synaptic neuron by simply using mod operator, however this is a memory optimisation rather than a

computation one and therefore does not affect our results. Our results demonstrate the need of careful neuron allocation based on connectivity and the inadequacy of round robin on large scale communication bound simulations.

In terms of scalability, the models tested in this chapter reach a lower computational density load per process as the level of parallelism is increased (since the model size is fixed, it reaches an average of 107 neurons per process in MVC simulated over 6,144 processes). Together with the increased communication overhead, this makes simulations markedly commmunication-bound. An argument could be made to attribute performance gains seen in figure 3.12 to an over-parallelisation of the simulation, resulting in a simulation that has an unrealistically low computation to communication. Nonetheless, figure 3.12F shows that with the same computation ratio (at any scale), our proposed *HP-NBX* clearly outperforms the alternative *round robin-PEX*, thus making it the preferred choice in high or low computation ratios. In larger, more complex simulations it could be the case that the computation ratio may be higher and therefore the simulation spends more time in computation; those cases are candidates for further parallelisation, in which our approach helps mitigate the poor scaling of communication requirements.

3.7 Cost of *HP-NBX* against the gains in simulation time

Performing hypergraph partitioning adds a cost to building the simulation that is calculated by taking the difference between build time for baseline and *HP-NBX*. Similarly, the time gain of the simulation is defined as the difference in simulation time between the baseline and *HP-NBX*, as a percentage of the build time. Figure 3.14 shows the build time and simulation time gain between *Round robin-PEX* and *HP-NBX* in 700ms of MVC simulations. Due to the use of a parallel implementation of hypergraph partitioning, build time difference (red in figure 3.14) is reduced when scaling. With lower simulation time (figure 3.12F), the runtime gain of *HP-NBX* reaches over 90% with 6144 processes. Taking the extra build time that *HP-NBX* requires, there is a total net gain (build and simulation time) for simulations run in 3072 and 6144 processes (denoted by time gains being higher than extra build time in figure 3.14).

3.8 Conclusions

Communication has been shown to be the dominant part of parallel SNN simulations as the number of processes is increased, limiting scalability. This chapter shows how to improve computational efficiency in distributed SNN simulations by optimising the three phases of communication: implicit synchonisation, process handshake and data exchange. We use neuronal connectivity to reduce inter-process communication, applying hypergraph partitioning to drive neuron allocation to processes. To tackle the impact of the mandatory handshake in point-to-point parallel communications, we implement *NBX* dynamic sparse strategy as a communication pattern.

With the use of a custom SNN simulator framework, which is representative of more general SNN simulators, this chapter demonstrates:

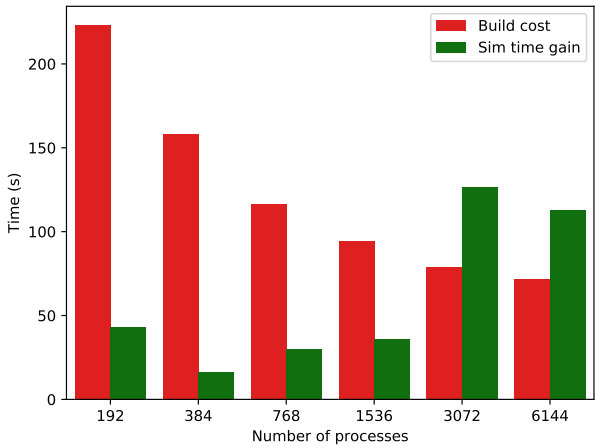


Figure 3.14: Comparing the cost of computing hypergraph partitioning in extra build time (in red) to the gains in simulation time by *HP-NBX* as a percentage of the build time (in green). As the simulation scales, the gains in simulation time compensate the extra build time associated with hypergraph partitioning.

- Hypergraph partitioning is shown to produce communication sparsity in distributed SNN simulations (up to 90% less ARN) and to reduce volume of data exchanged (up to 80% less volume of data) (contribution **C2**).
- Dynamic sparse communication *NBX* smooths process load imbalance introduced by *PEX*, resulting in reduced implicit synchronisation time by up to 90% (contribution **C3**).
- Synergy between partitioning and *NBX*: Hypergraph partitioning sparsity makes *NBX* more effective and reduces data exchange time. Hypergraph partitioning neuron allocation combined with *NBX* communication pattern increases computational efficiency by up to 40.8 percent points and reduces simulation time by up to 73%, compared to round robin allocation, a standard algorithm in neuronal simulators (contribution **C1**).

The findings are application agnostic and are applicable to other distributed complex system simulations in which communication can be modelled as a graph network.

3.9 Further work

This chapter evidences the benefits of using hypergraph models of SNN simulations and employ state-of-the-art partitioning algorithms to increase sparsity and performance on distributed simulations. However, current state-of-the-art hypergraph partitioning algorithms (multiscale) do not scale well [49, 125] and may incur in high memory requirements for very large graphs. Furthermore, those algorithms do not take hardware communication costs into consideration; with HPC systems being highly heterogeneous in nature, this is a potential source for optimisation.

Those limitations are addressed in further chapters of this thesis.

3.9.1 Out of scope work

In the context of neuromodulated plasticity, synapses weights are modified during learning influenced by synaptic activity and neuromodulator concentration in the brain region. Potjans et al. [178] show an efficient model to inform non-local synapses of the neuromodulator activity via a volume transmitter to reduce communication in distributed systems. Their model does not attempt to optimise this communication based on the locality of the synapses targeted. Including the volume transmitters in our hypergraph model had the potential to help reduce this communication.

It is conceivable that nature and evolution have shaped neuron populations to optimise communication, placing frequently communicating neurons closer together. Working at the neuron level we may be rediscovering this optimisations that nature has found. Comparing the partitions found via the hypegraph partitioning algorithm to actual biological structures based on proximity is an interesting area of future work.

A frequent alternative to optimise the synchronisation of parallel applications is to overlap communication and computation. The development of RDMA-capable hardware facilitates this —see section 2.3.2. An area of future work is to implement RDMA-enabled communication patterns and compare them to dynamic sparse data exchange.

We have assumed a fairly stable firing pattern across the SNN during our simulations. This is a simplification that has allowed us to study workload allocation and communication patterns in a simplified stable scenario. Natural SNN activity is more complex and dynamic and a model that attempts to optimise communication needs to reflect that. Repartitioning the hypergraph model based on actual activity of neurons can help model this complexity and improve runtime performance —see appendix A. Repartitioning requires faster online partitioning algorithms, which justifies the work in the next chapters.

Chapter 4

Architecture-Aware Restreaming to Improve Performance of distributed Applications Running on High Performance Computing Systems

4.1 Overview

High Performance Computing (HPC) demand is on the rise, particularly for large distributed computing. HPC systems have, by design, very heterogeneous communication capabilities consequence of their hierarchical architecture, resulting in wide variations in the cost of communications between compute units. Chapter 3 demonstrated the runtime performance benefits of using hypergraphs to model distributed applications and partitioning for workload allocation to minimise distributed communication. Chapter 2 summarised that current hypergraph partitioning algorithms do not consider heterogeneity when performing partitioning. It was suggested that this might impact the performance of distributed applications that are modelled as a hypergraph. If large distributed applications are to take full advantage of HPC, the physical communication capabilities must be taken into consideration when allocating workload. Strategies that take heterogeneous physical communication capabilities into account are considered to be *architecture-aware*. In contrast, strategies that do not account for this heterogeneity are referred to as *architecture-agnostic*.

Research question: can performance of distributed applications be improved by using architecture-aware hypergraph partitioning algorithms to allocate distributed workload in such a way as to reduce the cost of the necessary parallel communication?

When using hypergraph partitioning to drive workload in distributed applications in heterogeneous hardware, measuring the quality of the resulting partitions is not sufficient. The real cost of partitioning for a distributed application is determined not only by the number of hyperedges cut or the span of those cuts, but also by the real cost of communication between the computing nodes that host the elements within a hyperedge. Therefore it is necessary to develop appropriate benchmarks to analyse the impact of hypergraph partitioning algorithms on the runtime

of distributed applications.

To help answer the research question, this chapter proposes a synthetic distributed application runtime benchmark. The benchmark is required to address the limited availability of appropriate benchmarks summarised after reviewing the literature in Chapter 2. It simulates communication and computation needs of generic distributed applications that are modelled as hypergraphs.

4.1.1 Contributions

Hypergraphs are good at modelling total volume of communication in parallel and distributed applications. To the best of our knowledge, there are no hypergraph partitioning algorithms to date that are architecture-aware. In this chapter we propose a novel hypergraph partitioning algorithm (*HyperPRAW*) that takes advantage of peer to peer physical bandwidth to improve distributed applications performance in HPC systems. Physical bandwidth is estimated prior to partitioning through profiling. The results show that not only the quality of the partitions achieved by our algorithm is comparable with state-of-the-art multilevel partitioning, but that the runtime performance in a synthetic benchmark is significantly reduced in 10 hypergraph models tested, with speedup factors of up to 14x over architecture-agnostic state-of-the-art multilevel partitioning.

This chapter focuses on the benefits on runtime execution for distributed applications running on HPC systems when using architecture-aware partitioning. It does so by demonstrating performance improvement on a synthetic benchmark proposed in this chapter that models communication in distributed applications. The contributions of this chapter to the overall thesis are:

- **C5**: Demonstrate measurable runtime speedup (up to 14x) in distributed applications by allocating workload through a novel architecture-aware hypergraph partitioning algorithm.
- **C6**: Propose a novel cost function for restreaming hypergraph partitioning to balance heterogeneous communication costs and workload balance.
- **C7**: Develop a synthetic benchmark for modelling distributed applications in heterogeneous HPC systems to aid the evaluation of workload allocation strategies.
- **C8**: Improve the mapping of application communication patterns and hardware architecture heterogeneity by using a novel global communication metric (*partitioning communication cost*) that guides the refinement of restreaming partitioning.

4.2 Communication heterogeneity in HPC

In the world of Big Data and large scientific simulations, there is huge demand for High Performance Computing (HPC) systems. HPC systems achieve high performance through parallelism and distribution. By the distributed nature of their architectures, there is a level of communication heterogeneity between any two processes within nodes.

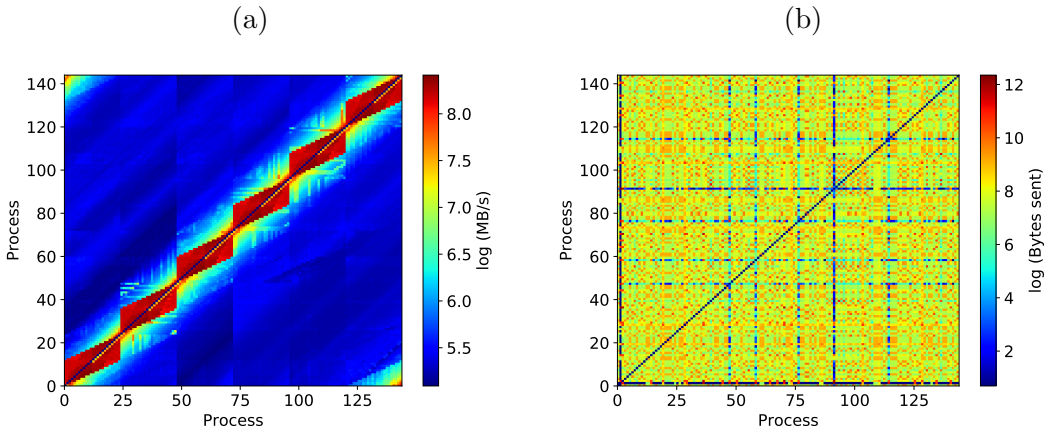


Figure 4.1: Discrepancies between network bandwidth in HPC systems and communication patterns in parallel applications. **A**: Peer to peer bandwidth heatmap on a 144 node job in ARCHER. **B**: Peer to peer communication activity pattern on a typical distributed application (run of our synthetic benchmark with *sparsine* hypergraph).

Take as an example the architecture of ARCHER[1], the UK National Supercomputer Service. Each compute node has two 12-core Intel Ivy Bridge processor. Four nodes are connected to an Aries router, 188 nodes are grouped into a cabinet; and two cabinets make up a group. There are all-to-all electric connections between nodes in the same group and all-to-all optical connections between different groups. This connectivity pattern, comparable to other HPC systems, leads to different connectivity speeds between computing units, depending on where they are hosted. Figure 4.1A shows the profiled bandwidth (real communication speed) between any two computing units within a cluster of 6 nodes in ARCHER (144 units), indicating substantial differences between processes communication bandwidth depending on where the processes are hosted. The graph closely represents the architecture of the system, with the highest speed connectivity between computing units within the same processor (red), followed by communication between the two processors in the same node (yellow, light blue)). All other connectivity is considerably slower (dark blue).

Data exchanged during a typical parallel application is shown in Figure 4.1B. The allocation of work is done with state-of-the-art hypergraph partitioning (Zoltan). The noisy pattern of the peer to peer data activity is a common feature of naive partitioning in which the total communication is optimised globally (total data sent over the entire network) but not at individual unit to unit link cost.

The mismatch between the network bandwidth pattern and the actual data sent during simulation (Figure 4.1) leads to uneven costs of communication. Since bandwidths between units are significantly different, the cost of sending data (in terms of time) is also different.

Even though profile results have been shown only for ARCHER, the findings are generalisable to any HPC systems, including cloud computing, due to their distributed architecture. Parallel applications running in HPC systems can improve their communication performance and overall runtimes by considering the network

[1] `http://archer.ac.uk/`

bandwidth of the architecture they run on to reduce the real cost of communication. Minimising the overall real cost of communication can be seen as an optimisation process and it is hypothesised here that automating it through architecture-aware strategies can lead to significant performance impact.

The goal of this chapter is to optimise distributed communication in HPC systems by reducing the mismatch observed in Figure 4.1. We propose a novel re-streaming hypergraph partitioning that is architecture-aware to reduce the real cost of communication in distributed applications.

4.3 A synthetic distributed application runtime benchmark

Hypergraph partitioning is used in domains such as VLSI circuit design and boolean satisfiability problems. Those are *one-shot* problems, once a solution has been found, there is no further problem. Hypergraphs can also model dynamic parallel applications to reduce total volume of communication in scientific simulations [73] and in sparse matrix multiplication [19]. In these cases, a good partitioning results in runtime improvements of the applications.

To benchmark communication in distributed applications, let us introduce the concept of *null-compute* simulations. A null-compute simulation is a model of a simulation that does not perform meaningful computation by either bypassing it or by using simple placeholders. They are often used to measure bottlenecks or performance impact that arise from specific parts of a computer system such as communication or memory overheads. Similar micro-benchmarks have been proposed as indirect estimates of communication time [87].

To measure the impact that our proposed strategy has on runtime communication of modelled distributed applications, we propose a synthetic benchmark. The benchmark is a null-compute simulation based on the input hypergraph and using a vertex allocation determined by the partitioning strategy selected. Since the simulation does not have any compute, it is purely communication-bound. The communication is proportional to hyperedge cut and SOED of the hypergraph and it is generated as follows: for each hyperedge on a given hypergraph, a message is sent to and from each vertex in the hyperedge if the vertices are located in different partitions (computing units). This is repeated for all hyperedges in the hypergraph. All communication is performed with discrete point-to-point[2] messages using the MPI library for distributed systems communication. Figure 4.2 highlights the communication required (arrows) for a simple hypergraph partitioned across three nodes (different colours).

Although the synthetic benchmark is an extreme case of parallel application (no compute and communication of all connected compute elements on every time step) it abstracts a communication-bound distributed application (such as certain Spiking Neuronal Network simulations [73]). It allows us to compare the impact of different partitioning algorithms and understand how they improve communication in communication-bound distributed applications, in which scaling is limited by the overheads imposed with increased communication. A general benchmark allows cross-domain applications to be evaluated, from matrix multiplication to SNN

[2]MPI_Send and MMPI_Recv calls from sending and receiving processes respectively.

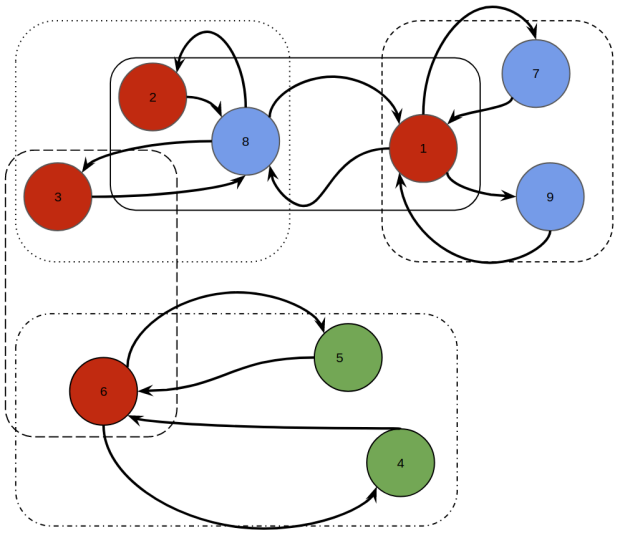


Figure 4.2: Synthetic benchmark proposed to measure the impact that hypergraph partitioning workload allocation has over distributed applications. The diagram shows a hypergraph with 9 vertices (circles) allocated to three different computing nodes / partitions (red, blue and green). Hyperedges connecting vertices are shown as dashed rectangles. Communication in runtime (arrows) happens between vertices belonging to the same hyperedge but allocated to different partitions (i.e. vertices within the same rectangle but of different colours).

simulation to circuit design, which would not be possible if we were to use a domain-specific framework.

To account for variable network traffic and different nodes configurations provided by the job scheduler, the runtime experiments are run on three different jobs (hence different node placement and communication costs), with each job doing two iterations. Therefore the total number of simulations run per experiment is 6.

4.4 Restreaming partitioning to improve distributed application performance

Previous work on graph partitioning has already highlighted the impact that uneven computation and communication architectures in HPC and Cloud computing has on computation performance [223, 225]. A hypergraph (a generalisation of graphs where edges can link n number of vertices) is shown to model total volume of communication in parallel applications [44, 63, 61]. Once the application is modelled as a hypergraph, partitioning algorithms can be used to reduce the communication volume. Chapter 3 demonstrates how hypergraph partitioning improves the performance of SNN simulations.

Partitioning algorithms for hypergraphs with good quality results exist using a variety of algorithms —see section 2.7.2 for an overview. However, as discussed in section 2.9.1, there is currently no approach that considers the physical architecture of the network to partition and allocate hypergraph models. When modelling parallel applications as a hypergraph, not only it is important to reduce the hyperedge cut (connection between two vertices located in two different partitions), but also

the hop cut (connections including the physical cost of communication).

Parallel applications may not have constant communication patterns across their runtime. Using static approaches ahead of execution to distribute workload may not yield the best results in those circumstances. Repartitioning algorithms (those that perform the partitioning more than once) consider this and previous work proposes to model the cost of migrating data [46, 47] as part of the partitioning, in addition to cut minimisation.

Section 2.8.2 reviews the difficulties of partitioning and repartitioning algorithms at large scale. Multilevel partitioning algorithms run into problems with large scale graphs [49, 125], since they require the graph to be fully loaded in memory to be processed. This imposes excessive memory requirements in the era of social media graphs and whole brain simulations, where real applications must handle billions of nodes and trillions of edges ([122]). Furthermore, even if graphs can be stored in memory, having to process the entire graph before allocating vertices to nodes imposes practical limitations such as long partitioning times.

Streaming and restreaming approaches do not suffer from the limitations expressed above: they can make allocation decisions with only local information (not needing the entire graph in memory) which means they can work with partially known graphs. This makes streaming algorithms ideal for large-scale graphs, which is the case in problems such as large neuronal simulations or multiplication of very large sparse matrices.

Figure 4.3 depicts the streaming process for the case of hypergraph partitioning. Figure 4.3A represents the input hypergraph, whereas Figure 4.3B models the current workload of each of the partitions based on current allocations. Figure 4.3C represents the streaming process, where one vertex is considered at a time. Based on local information, a value function is calculated per partition and the vertex is assigned to the one with higher value. The parameters included in the value function determine what characteristics the partitioning algorithm prioritises. To date, no streaming hypergraph partitioning algorithm considers network communication (bandwidth or latency) in the value function.

4.4.1 Architecture-aware streaming hypergraph partitioning

A hypergraph is a generalisation of a graph $H = (V, E)$ consisting of a set of vertices V and a set of hyperedges E, where each hyperedge is a subset of V (one or more vertices) that defines the connectivity pattern. The difference between graphs and hypergraphs is that in the latter the cardinality (size) of edges is always 2. For a formal definition, see section 2.7. Hypergraph partitioning is a process that assigns vertices to partitions in such a way that the hyperedge cut or the fanout metric is minimised. To avoid trivial solutions (such as assigning all vertices to one partition) partitioning algorithms maintain load balancing by only allowing solutions that have a total imbalance factor that is below a specified value (a tolerance imbalance).

Hypergraphs are good at modelling parallel communication when each hyperedge represents a frequent communication group of vertices. The more a hyperedge is cut (more partitions are involved) the more the modelled application will have to send data across partitions and hence more communication is required. When using hypergraphs to model parallel applications, the goal is to partition the hypergraph

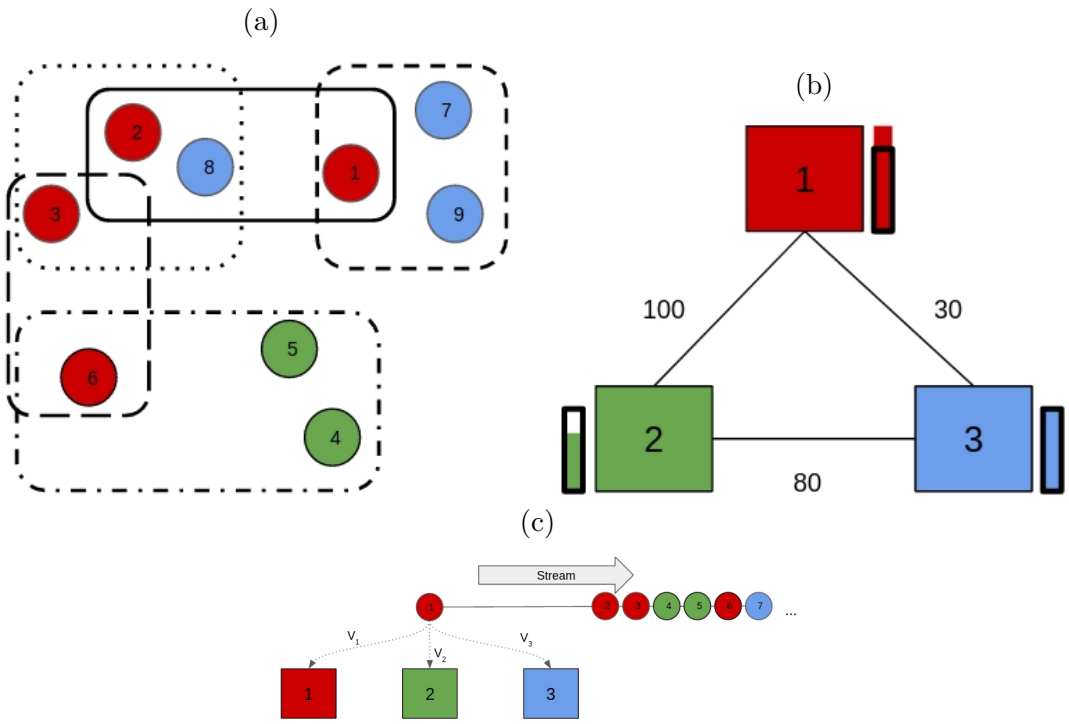


Figure 4.3: Overview of a hypergraph streaming partitioning process. **A**: Arbitrary hypergraph with vertices coloured based on the partition they are currently assigned to and hyperedges represented by dotted rectangles. **B**: model of the architecture and the current workload allocation; each box represents a partition (a compute unit in the architecture), with links weighed based on the physical peer to peer bandwidth of the architecture; an adjacent vertical bar represents whether the partition is currently overloaded or underloaded (measure of imbalance). **C**: Streaming process in which one vertex at a time is considered and assigned to a partition based on local information.

in k partitions, where each partition represents a computing unit in the hardware architecture the application runs on.

A streaming graph partitioner algorithm does not have the entire graph in view when calculating the cost of allocating a vertex to a partition. They are frequently called greedy since once they make a decision, it is not revoked later on after seeing more vertices. When an algorithm applies more than one pass (repeats the stream that visits the vertices once), it is often referred to as a restreaming approach. Our proposed algorithm takes a similar approach to the graph restreaming software GRaSP [21] but it is applied to hypergraphs. To keep a good balance between the two opposing goals (workload balance and minimisation of total communication) we use a tempering parameter α that weighs the importance of workload imbalance. In the streaming partitioning algorithm FENNEL, [212] suggest a good starting value for α that scales with other hypergraph features:

$\alpha = \sqrt(p) \times \frac{|E|}{\sqrt{|V|}}$,

where p is the fraction of partitions, $|E|$ is the number of hyperedges and $|V|$ is the number of vertices. After each stream this α value is updated (the update parameter is set to 1.7). Our approach differs from GRaSP in two ways: the restreaming is allowed to continue until the partition is no longer improved (what we call the refinement phase) instead of stopping once the imbalance tolerance is reached; we reverse the tempering of the workload imbalance weigh once we are within imbalance tolerance—see section 4.7.1. These two key differences allow for a refinement of the quality of the partition after reaching workload imbalance tolerance.

4.5 Proposed architecture-aware restreaming algorithm

The proposed architecture-aware restreaming algorithm (*HyperPRAW*) is described in Algorithm 1. The key step is the assignment of a vertex to a partition based on a value function. The partition that ends up with the highest value is the one to which the algorithm assigns the vertex.

The algorithm has a computational complexity that grows with the number of iterations (N), the number of vertices ($|V|$) and hyperedges ($|E|$), the hyperedge cardinality and the number of partitions (p). The implementation of HyperPRAW can be found on the GitHub repository at `https://github.com/cfmusoles/hyperPraw`

The next section describes what goes into calculating the value of assigning any vertex to each partition.

4.5.1 Vertex assignment cost function

To incorporate the harware communication costs, a novel vertex cost function for streaming partitioning is proposed. The value function $V_i(v)$ in equation 4.1 determines the value associated with assigning vertex v to partition i.

$$V_i(v) = -N_i(v) \times T_i(v) - \alpha \frac{W(i)}{E(i)} \tag{4.1}$$

where $N_i(v)$ represents the number of partitions in which vertex v has neighbouring vertices (described by equation 4.2), $T_i(v)$ is the total cost of communication

Algorithm 1: Sequential *HyperPRAW*: architecture-aware restreaming algorithm

Input : p (number of partitions); α (starting workload balance weight); t_α (workload balance tempering parameter); *imbalance_tolerance* (maximum imbalance tolerance); N (maximum iterations)

Output : P_k, for $k = 1, ..., p$, where P_k is the subset of V that is allocated to partition k

Data : $H = (V, E)$, where V is a set of vertices and E is a set of hyperedges for hypergraph H

Initialise P_k: Round robin assignment of each $v \in V$
Calculate $W(k)$ for each partition k
for $n = 1$ *to* N **do**
    for $v \in V$ **do**
        $j \leftarrow \arg\max_{k=1,...,p} = -N_k(v) \times T_k(v) - \alpha \frac{W(k)}{E(k)}$
        Add v to set P_j
        Recalculate $W(j)$
    $\alpha \leftarrow \alpha \times t_\alpha$
    if *imbalance* > *imbalance_tolerance* **then**
        continue
    else if *Cost of* P_k^n > *Cost of* P_k^{n-1} **then**
        return P_k^{n-1}
    else
        continue
return P_k^N

due to assigning vertex v to partition i (described in equation 4.4), $W(i)$ is the current workload of partition i and $E(i)$ is the expected workload for partition i. The parameter α weighs the importance of workload balance in the overall cost. Since it starts at a low value, the initial streams partition mostly based on communication cost. At later streams, the workload balance gains importance to achieve balanced partitions.

Throughout our experiments we have assumed even cost of computation per vertex and homogeneous work capacity for partitions, hence assigning one vertex to one partition increases by 1 its workload, and the expected workload for all partitions is the total workload divided the number of partitions. However the algorithm can easily account for heterogeneous computation and work capacities.

$$N_i(v) = \frac{\sum_{j=0}^{p} A_j(v)}{p} \tag{4.2}$$

$$A_j(v) = \begin{cases} 1, & \text{if } X_j(v) > 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

The function $N_i(v)$ indicates the number of neighbouring partitions of vertex v, should it be placed in partition i. That is, in how many other partitions v has connecting vertices ($A_j(v)$ indicates whether vertex v has neighbours in partition j).

$$T_i(v) = \sum_{j=0}^{p} X_j(v) \times C(i,j) \tag{4.4}$$

The function $T_i(v)$ computes the cost of communication associated with allocating vertex v to partition i. That cost, for another partition j is the number of neighbours v has in j ($X_j(v)$) multiplied by the cost of communication between partitions i and j ($C(i,j)$ which is discussed in section 4.5.2). The total communication cost is the sum of all costs for each partition other than the one v is assigned to (i) —communication within a single node is considered *free*.

In order to successfully calculate the total cost due to communication, the algorithm requires information regarding the cost of communicating between partitions (i.e., computing units). The next section describes how to map this cost.

4.5.2 Mapping cost of communication

The cost of communication matrix is derived from the peer to peer bandwidth calculated through profiling before the partitioning starts the streaming process. Discovery through profiling gives the flexibility as it can be applied to any architecture topology and it will discover the network costs automatically. This is an advantage in environments where the architecture is not known (in Cloud computing), or when it is known but unreliable due to contextual circumstances (shared network resources). If the bandwidth or communication costs are known, they can be used directly without the need for profiling. Profiling is a common method to discover the physical topology graph in partitioning algorithms [157, 23, 181].

When the cost of communication is done through profiling, first the bandwidth matrix is found, then it is transformed to represent cost of communication and not

speed. The bandwidth profiling is done by iteratively sending data to and from MPI processes arranged in a ring formation[3] and timing how long it takes for them to reach back. One MPI process per core is used.

With the peer to peer bandwidth data we calculate the cost of communication in the following way:

$$C(i,j) = 2 - \frac{b_{ij} - b_{min}}{b_{max} - b_{min}} \tag{4.5}$$

where i and j represent two cores, b_{ij} is the bandwidth between core i and core j, and b_{min} and b_{max} are the maximum and minimum bandwidth between any two cores in the network. This normalises the costs to 1 for the fastest link, and 2 for the slowest. When $i == j$, $C(i,j) = 0$. The normalisation step helps the streaming to be independent of the magnitude of bandwidth values. Since different hardware architectures can have different orders of magnitude bandwidths, the magnitude affects the balance between workload and communication cost used in the vertex assignment function (equation 1), potentially resulting in slower performance (if the cost values are too high) or sub-optimal solutions (if the cost values are too low, the stream can end underestimating the communication cost)

To accurately model the underlying architecture, the cost matrix must be calculated every time a new allocation of computing nodes is presented. In typical HPC jobs, this requires us to profile the architecture of the allocated cluster of nodes each time a job is started (since potentially new nodes are given).

4.5.3 Monitoring metrics during refinement

To improve the quality of partitioning, we propose a refinement phase to the restreaming algorithm after the workload imbalance has reached values below the desired imbalance tolerance. During the refinement phase, the restreaming continues (i.e., further iterations are run) until a monitored quality metric ceases to improve. A novel metric used is proposed: the *partitioning communication cost*. For a partitioning P, the partitioning communication cost $PC(P)$ is:

$$PC(P) = \sum_{i=0}^{k} \sum_{v} T_i(v), \text{ for all } v \in P_i \tag{4.6}$$

This uses the cost of communication $T_i(v)$ in equation 4.4 for all vertices and any partition i and aggregates it. Intuitively, this metric measures both the number of neighbours per vertex that live in different partitions to the vertex and the cost of communication between those partitions. This closely represents the volume and cost of communication in parallel applications that can be modelled with a hypergraph.

[3]The tool available in `https://github.com/LLNL/mpiGraph` is used for profiling bandwidth

Table 4.1: Hypergraphs used in quality and runtime evaluation

Hypergraph	Vertices	Hyperedges	Non zeros	Avg cardinality	H/V
sat14 itox dual	441729	152256	1143974	7.51	0.34
2cubes sphere	101492	101492	1647264	16.23	1.00
ABACUS shell hd	23412	23412	218484	9.33	1.00
sparsine	50000	50000	1548988	30.98	1.00
pdb1HYS	36417	36417	4344765	119.31	1.00
sat14 10pipe primal	77639	2082017	6164595	2.96	26.82
sat14 E02F22	27148	1301188	11462079	8.81	47.93
webbase-1M	1000005	1000005	3105536	3.11	1.00
ship 001	34920	34920	4644230	133	1.00
sat14 atco dual	561784	59517	2167217	36.41	0.11

4.6 Experimental evaluation

4.6.1 Experimental design

To evaluate the performance of the architecture-aware streaming, a public dataset[4] [193] is used. It includes a wide collection of hypergraphs used in various competitions (routability placement, circuit benchmark, SAT competition) and sparse matrices repositories. To test our approach we have selected 10 instances from within this collection that range in size, average cardinality and ratio number of hyperedge/vertex—see table 4.1.

The evaluation consists on running hypergraph partitioning algorithms over a collection of hypergraphs and measuring the partition quality and runtime on the proposed synthetic benchmark —see section 4.3. For both quality and runtime experiments we use a state-of-the-art multilevel recursive bisection partitioning algorithm (*Zoltan* implementation [62]) as a baseline. To understand the impact of using the physical architecture cost of communication in our restreaming approach we use two versions of the proposed partitioning algorithm: *HyperPRAW-basic* (where uniform cost of communication matrix is used) and *HyperPRAW-aware* (where cost of communication matrix from bandwidth profiling is used).

Three experiments are carried out: the impact of refinement in restreaming (in section 4.7.1), resulting partitioning quality evaluation (in section 4.7.2), and runtime performance of hypergraphs on the synthetic benchmark (in section 4.7.3). The experiments are designed to help assess the impact of the proposed architecture-aware restreaming algorithm on modelling distributed applications:

- Demonstrate the importance of incorporating communication costs when using hypergraphs as models of distributed applications by baseline performance with an architecture-agnostic hypergraph partitioning (Zoltan).
- Show the isolated impact of incorporating architecture-awareness in the restreaming algorithm by comparing bandwidth data through profiling versus predetermined uniform bandwidth cost.

[4]Dataset is accessible via Zenodo at `https://zenodo.org/record/291466`

- Demonstrate that Partitioning Communication cost (PC) is good as a proxy for total application communication by using it as the metric to optimise during refinement.

All experiments are run in ARCHER, the UK National Supercomputing system. To ensure there is enough architecture communication heterogeneity, the job size is set to 576 cores. With 24 computing units per computing node this ensures multiple distributed nodes (24) are employed across physically distinct (6) blades —see section 3.4.2 for a detailed description of ARCHER's architecture.

4.6.2 Quality and runtime metrics

Hypergraph partitioning algorithms traditionally optimise one of the two metrics: hyperedges cut (number of hyperedges that contain vertices that are allocated to more than one partition); Sum Of External Degrees (for each partition, sum of the hyperedges that are incident on the partition but not fully contained in it).

Formally, the Sum Of External Degrees (SOED) is $\sum_{i=0}^{k} |E(P_i)|$ for k partitions, where $E(P_i)$ is the number of hyperedges that are incident but not fully inside partition i. Intuitively, high values of SOED indicate hypergraphs are being cut across several partitions, representing more volume of communication.

Both hyperedge cut and SOED are cut-based metrics (calculated on the basis of hyperedges cut across partitions) and give an indication of the static quality of the partition. They are used in this work to report the quality of *HyperPRAW*. In addition, the PC cost defined in equation 4.6 is also used as a metric that combines cut information and physical cost of communication.

Quality of hypergraph partition only describes the results on the hypergraph itself. The hypergraph in this work is used as a model for a parallel application to improve performance. To measure the improvement that *HyperPRAW* can bring to these parallel applications, we use time execution on a synthetic benchmark to measure communication time during simulated runtime —see section 4.3.

4.7 Results

4.7.1 Refinement phase

To understand the effect of the refining phase, we compare the partition history of *HyperPRAW* for alternative stopping conditions of the restreaming process: no refinement (stop restreaming when the imbalance tolerance has been reached), refinement 1.0 and refinement 0.95 (continue restreaming until the partitioning quality is no longer improved). The refinement value determines the update of the tempering parameter used once the imbalance tolerance is reached (1.0 results in the α parameter not being updated, whereas 0.95 decreases the value of α, and hence the importance of workload imbalance, instead of increasing it).

Figure 4.4 shows the partitioning history for 4 hypergraphs. The figure demonstrates how partitioning communication cost decreases with more iterations; this is the metric that is used to monitor and stop refinement and directly correlates with the amount of communication modelled by the hypergraph. Comparatively, both refinement strategies perform better than not refining at all. Using an update value

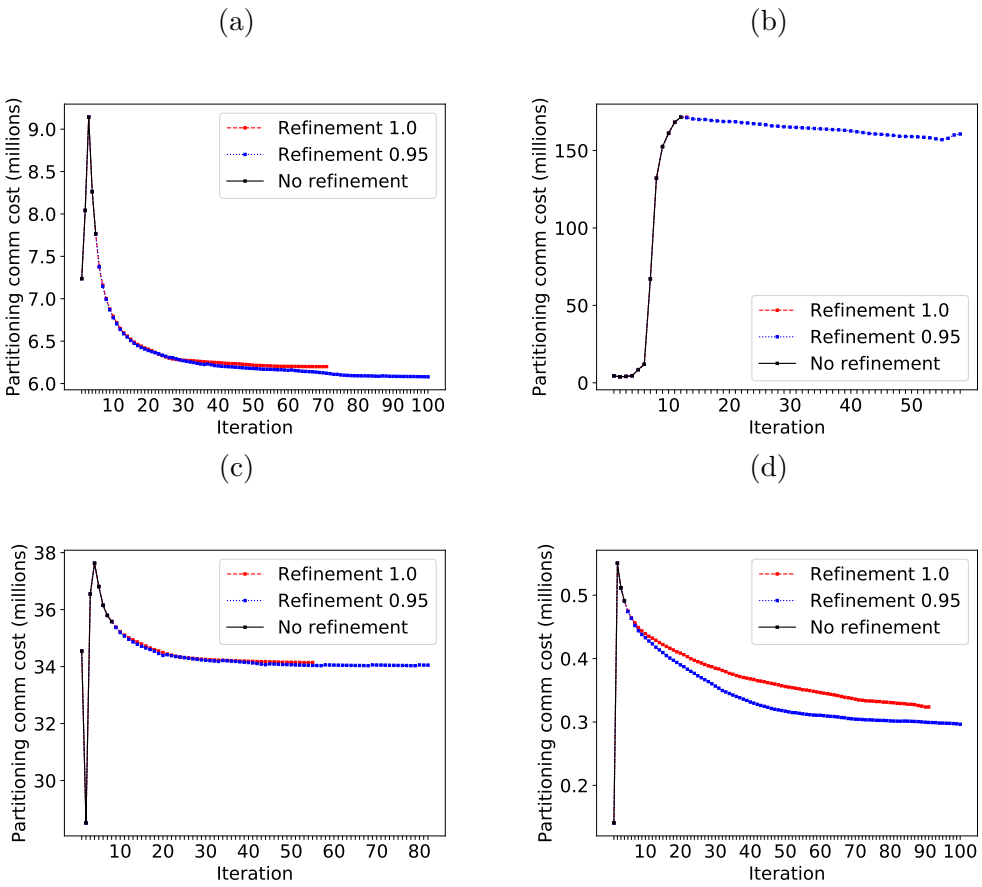


Figure 4.4: Partition history of the *HyperPRAW* algorithm comparing different refinement strategies: no refinement (black), refinement 1.0 (red dashed) and refinement 0.95 (blue dotted). **A**: *2cubes sphere* hypergraph. **B**: *sat14 itox vc1130 dual* hypergraph. **C**: *sparsine* hypergraph. **D**: *ABACUS shell hd* hypergraph.

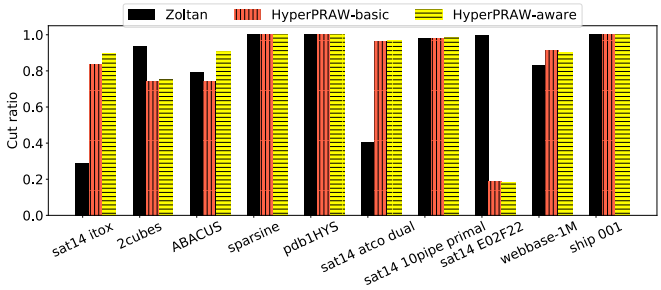
for the tempering parameter that decreases the importance of workload balance (by a factor of 0.95 at each iteration) reaches the lowest levels of partitioning communication cost, therefore improving the restreaming quality. The initial worsening of the outcome of the partition, indicated by a peak of the Partitioning communication cost (PC) in the earlier iterations is due to the algorithm settling on partitions that meet imbalance tolerance criteria, as often the α parameter starts low enough to encourage solutions that place most vertices within the same partition —hence with low PC.

4.7.2 Quality of partitioning

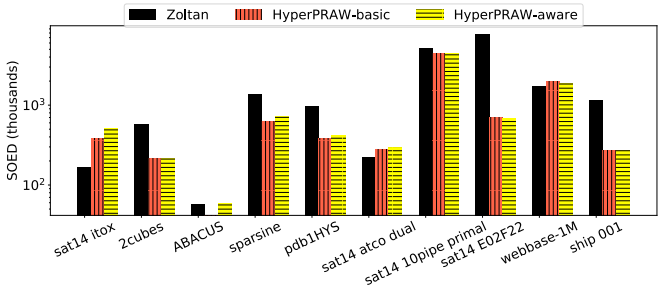
The quality of the partitioning of both versions of *HyperPRAW* and *Zoltan* is shown in Figure 4.5. In terms of standard hyperedge cut, *HyperPRAW* shows results that are below but comparable to *Zoltan* (from the 10 hypergraphs, the hyperedge cut is worse in 4, better in 2 and about the same in the other 4). When measuring the SOED, a metric that better models total volume of communication in parallel applications, the results are slightly better for *HyperPRAW* (3 worse instances, 1 about the same and 6 where it is better).

Neither the SOED nor the hyperedge cut include the physical cost of communi-

(a)



(b)



(c)

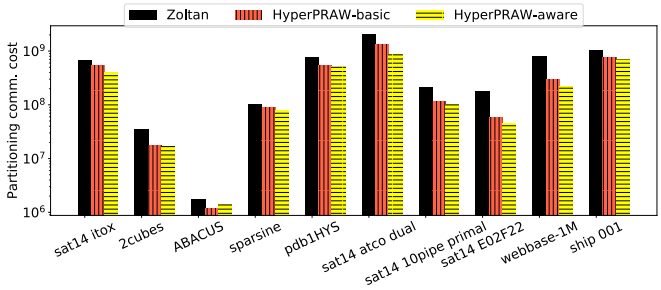


Figure 4.5: Quality metrics on 10 hypergraphs comparing the partitioning algorithms: *Zoltan* (black), *HyperPRAW-basic* (orange vertical lines) and *HyperPRAW-aware* (yellow horizontal lines). **A**: Hyperedge cut. **B**: Sum of External Degrees (SOED) in logarithmic scale. **C**: PC in logarithmic scale.

cation. The PC metric considers it and it is where *HyperPRAW* obtains the best results, with improvements over *Zoltan* on all hypergraphs and *HyperPRAW-aware* outperforming the basic alternative. Note that *HyperPRAW-aware* is the only one that uses the real cost of communication matrix during the partitioning. Both *Zoltan* and *HyperPRAW-basic* assume uniform costs and only make use of the physical cost of communication to calculate the final partitioning cost displayed in the figure.

4.7.3 Runtime performance on benchmark

Figure 4.6 shows the overall runtime for 10 hypergraphs on our synthetic benchmark. The results show that *HyperPRAW-basic* reduces the simulation runtime with respect to *Zoltan* and *HyperPRAW-aware* further improves that significantly. The speed up factors of *HyperPRAW-aware* over *Zoltan* range from 1.3x to 14x.

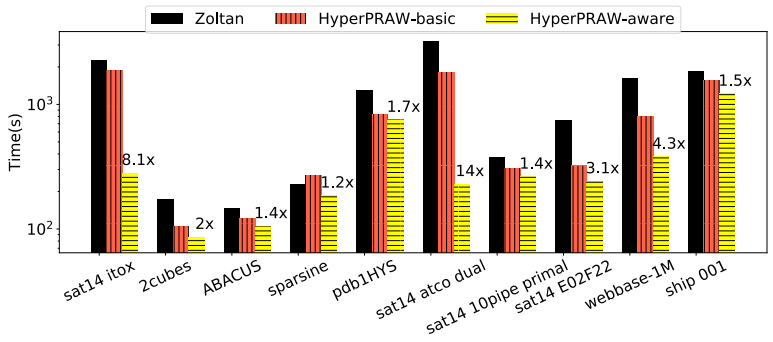


Figure 4.6: Runtime performance (in logarithmic scale) on a synthetic benchmark for 10 hypergraphs comparing the partitioning algorithms: *Zoltan* (black), *HyperPRAW-basic* (orange vertical lines) and *HyperPRAW-aware* (yellow horizontal lines). The speedup factors of *HyperPRAW-aware* over *Zoltan* are annotated in the figure.

4.8 Discussion

In other restreaming partitioning algorithms, the iterative streams are halted when the workload imbalance tolerance is reached [21]. Nonetheless it is possible that the partitioning quality could improve if streams are allowed to continue despite being within acceptable imbalance. Figure 4.4 demonstrates the effectiveness of a refinement phase, where the streams continue until a partitioning metric ceases to be improved. Although the restreaming goes for longer (more iterations), this results in a higher quality partition (as measured by the PC, a suitable metric for hypergraphs that model parallel communication). The best alternative is found to be when during the refinement phase, the workload imbalance weight parameter α is reduced (refinement 0.95), instead of increased as it is done when outside of workload imbalance tolerance (update value of 1.7). The value 0.95 was found experimentally, since lower values may force the algorithm to fluctuate in and out of the acceptable load balance range, degrading performance. Nonetheless, the intuition is that the best partition is found when the balance constraints are relaxed (an extreme case is when all workload is assigned to one partition; there is no cut or communication in such an assignment, but we have maximum imbalance). Once the algorithm is within the range of acceptable partitions (i.e., within the imbalance tolerance), we can search for slightly more imbalanced solutions in an attempt to find an acceptable solution that is maximally imbalanced.

Figure 4.4 also shows that the effectiveness of the refinement phase varies with the hypergraph, indicating that some types of hypergraphs do benefit more from a refinement than others. The refinement factor is therefore a candidate parameter to tune empirically.

Figures 4.5A and 4.5B show the cut-based metrics. In hyperedge cut, *HyperPRAW* underperforms *Zoltan* in 4 of the 10 hypergraphs, but in SOED the *Zoltan* benchmark is outperformed in 6 hypergraphs, resulting in an overall comparable performance. When the physical communication cost matrix is considered, as is the case with PC (Figure 4.5C), the quality metric shows *HyperPRAW* being consistently superior to *Zoltan* in all hypergraphs. This work attempts to improve the performance of distributed applications, for which we model the application as a

hypergraph that is partitioned using our proposed restreaming approach. The output from the restreaming algorithm is a scheme that is used to distribute workload in a heterogeneous environment and its quality is ultimately evaluated indirectly by timing how long the application runs for under the new scheme. Therefore our focus is on end runtime performance improvement, and although we report traditional partitioning quality metrics, this is done descriptively and not as a target metric. This is the reason to report the partition quality with the PC, since it reflects more accurately the needs of the target to optimise (peer to peer communication in a heterogeneous environment). Therefore, the resulting quality of the restreaming partitioning shown in Figure 4.5C indicates a net improvement over *Zoltan*.

The runtime performance on the synthetic benchmark in Figure 4.6 confirms the results obtained in the PC metric. In 9 out of 10 hypergraphs, both versions of *HyperPRAW* outperform *Zoltan*, showing the effectiveness of the proposed restreaming approach. The results also show that the restreaming approach benefits from using the physical communication cost matrix, where in all cases *HyperPRAW-aware* achieves faster simulation times than the basic counterpart and *Zoltan*. When compared with *Zoltan*, *HyperPRAW-aware* reaches significant speedup factors ranging from 1.3x to 14x (with 3 hypergraphs reaching speedups above 4x).

A paradigmatic case of the importance of the PC metric over the cut-based ones is found on two hypergraphs: *sat14_itox_vc1130_dual* and *sat14_atco_enc1_opt1*. In both cases, the hyperedge cut and the SOED metrics are worse on the restreaming approach than in *Zoltan*. However, the PC is better in *HyperPRAW*, with an outstanding runtime speedup on the synthetic benchmark of 8.1x and 14x respectively.

HyperPRAW relies on the information built through profiling to construct the communication cost matrix. Using a simple MPI send-receive ring protocol is empirically seen to be sufficient to successfully map the known hardware topology in ARCHER. Figures 4.1A and 4.7A show the characteristic 24 process clusters of high speed communication in ARCHER, which map to cores within a single computing node. Within a 24 process cluster we also see two tiers, corresponding to the two 12 cores Intel Ivy Bridge processors.

Earlier we evidenced the issue of running parallel applications in heterogeneous HPC systems by profiling the communication pattern of distributed application and the peer to peer bandwidth of a group of computing units.. Figure 4.7A shows the peer to peer bandwidth for a job allocation in ARCHER where we run the synthetic benchmark. As expected from its architecture, the fastest communication links are on neighbouring units (each group of 24 units belonging to the same computing node), thus the pattern of high bandwidth in the central band. For an optimal utilisation of the hardware architecture, the patterns of activity of the parallel application should resemble that of the bandwidth profile. This is what we see when showing the pattern of activity of our synthetic benchmark for sparsine hypergraph partitioned with *Zoltan* (4.7B), *HyperPRAW-basic* (4.7C) and *HyperPRAW-aware* (4.7D). For the first two, since they do not use the physical communication cost matrix, the pattern of communication is uniformly random. However, for *HyperPRAW-aware*, using the communication cost matrix makes the restreaming distribute the communication pattern to closely resemble the peer to peer bandwidth. Therefore our approach is able to better exploit fast interconnections between computing units.

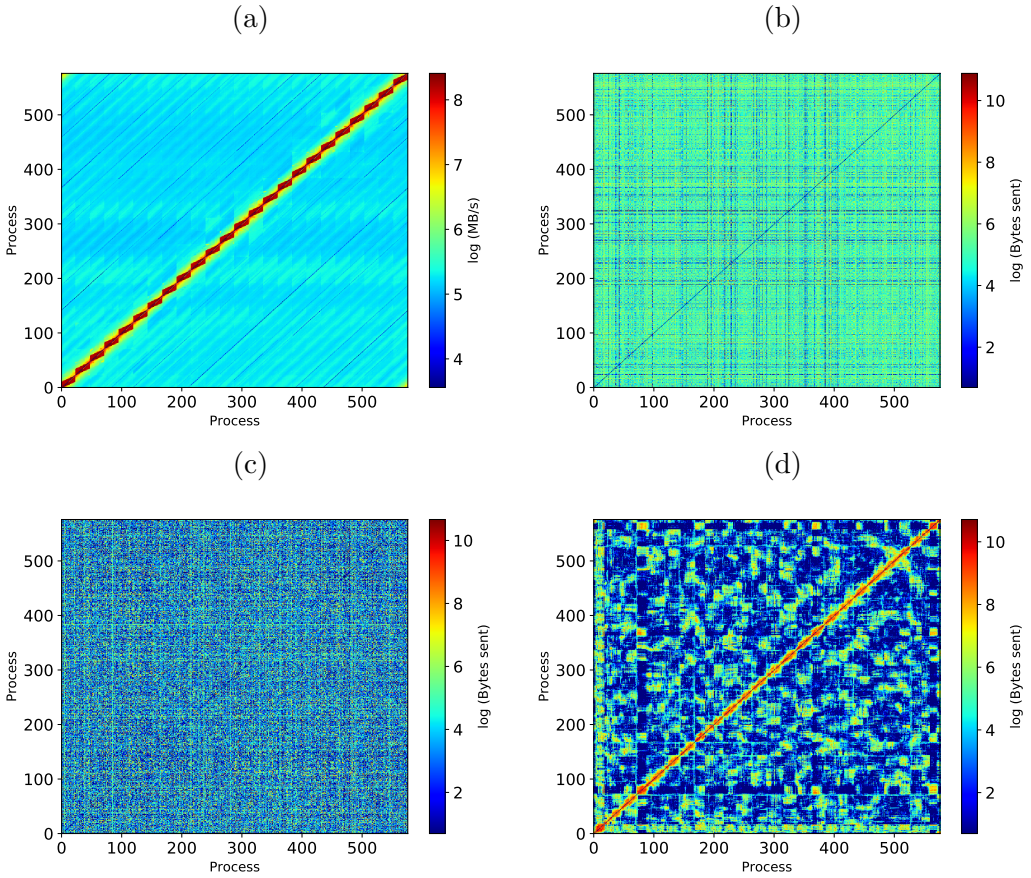


Figure 4.7: Architecture bandwidth compared to peer to peer communication pattern on the synthetic benchmark. **A**: peer to peer bandwidth of a 576 computing units job in ARCHER. The bottom part of the figure represents the peer to peer communication pattern on the synthetic benchmark run of the sparsine hypergraph: **B**: communication pattern after using *Zoltan*; **C**: Communication pattern using *HyperPRAW-basic*; **D**: Communication pattern using *HyperPRAW-aware*.

4.8.1 Design of the allocation cost function

In the proposed allocation cost function shown in equation 4.1, $N_i(v)$ and $T_i(v)$ play complementary roles. $N_i(v)$ measures the future external degree of the hyperedge, or in how many partitions the hyperedge has a presence (i.e. vertices in them). $N_i(v)$ is therefore designed to penalise allocations that spread hyperedges across partitions, but would not mind assigning further vertices to a partition where the hyperedge is already present. $T_i(v)$ measures instead the weighted external degrees using both the number of vertices present on each partition per hyperedge as well as the communication cost between partitions. $T_i(v)$ is designed to penalise adding vertices to partitions with slow communication links between partitions where the hyperedge is present, but often may result in adding a vertex in a partition where the hyperedge is not present as long as the communication cost between existing and new partition is fast enough. Combining both results in a good balance of both.

To make the algorithm architecture-aware, communication costs are estimated indirectly via profiling. Profiling helps discover the bandwidth between computing nodes (higher is better). First, the bandwidth value is normalised to remove the influence of magnitude on the overall cost function; this helps the algorithm not to overwhelm other parameters (such as workload imbalance). After normalisation, bandwidth still needs to be converted to cost (lower is better), by performing linear inversion (2 minus the normalised bandwidth). The rationale for choosing linear inversion is to keep relative distances between bandwidths uniform independent of the value of bandwidth (linear transformation). Alternative bandwidth to cost conversions may include inverting the value (1 over the normalised bandwidth), which would penalise more having lower bandwidth. A comparison of the impact of bandwidth to cost conversion is identified and left as future work.

4.8.2 Related work

Previous work on architecture-aware graph partitioning can be divided according to their partitioning strategy: local improvement or refinement with greedy strategies considering communication costs ([234, 161, 154]); streaming greedy partitioning with communication cost as part of the allocation function ([225, 223]); and synchronous partitioning of the machine graph and the application graph [48]. Although these approaches are indirectly applicable to hypergraphs by converting the model to a graph version, it has been shown that doing so cannot be done without trade offs, such as increase in graph size and edge explosion [117, 122].

Zoltan [62] offers a hierarchical approach for partitioning a hypergraph. The focus is on being able to use high cost algorithms at levels where reducing communication is more important and low cost ones when the communication may not impact as much. However, this approach only establishes qualitatively differences between architecture levels and does not model well the cost of communication between computing units belonging to different hierarchies. This approach is not easily applicable in environments where the architecture is not know directly (in Cloud computing) or it is known but unreliable due to contextual circumstances (e.g. shared network resources).

An alternative to optimise network communication in parallel applications is achieved via topology mapping —see section 2.3.3 for an overview. *LibTopoMap* [112] maps MPI processes to arbitrary network topologies to direct high communi-

cating processes to high bandwidth links. Note that this strategy does not redistribute work to minimise communication but rather maps the existing application parts (application processes or jobs) to a computing node network. Our approach has a finer level of granularity, since it is able to create the application partitions to minimise inter-partition communication, as well as mapping that communication to faster network links.

4.9 Conclusion

This chapter demonstrates the importance of architecture-awareness for distribution of workload in HPC systems in parallel applications. We propose *HyperPRAW*, an architecture-aware restreaming partitioning algorithm that optimises communication by understanding the underlying network bandwidth. The contributions of this chapter are:

- Incorporating physical cost of network communication to streaming partitioning has been shown to significantly improve runtime simulation on a synthetic benchmark, with up to 14x compared to architecture-agnostic strategies (contribution C5).
- Experimental validation has shown that specifically architecture-awareness can be attributed to most of the performance improvement (contribution C6).
- Propose a novel synthetic benchmark to aid distributed application runtime communication and computation modelling using hypergraphs (contribution C7).
- Using a novel global communication metric (PC) to guide the partitioning refinement phase (restreaming) is shown to result in reduced communication time in runtime benchmark (contribution C8).

4.10 Further work

This chapter has focused on the performance gains of parallel and distributed applications that an architecture-aware partitioning approach provides. One limitation of this work is that the restreaming partitioning is performed sequentially on a single processor. This limits the applicability to larger scale problems (high number of partitions and larger hypergraphs), both in terms of speed of execution and memory requirements —currently requiring the entire hypergraph to fit in memory. This limitation could be mitigated if the restreaming algorithm is adapted to work in parallel. Parallelisation of the streaming partitioning algorithm is explored in chapter 6, as well as other performance aspects such as memory requirements (removing the need to load the entire graph in memory) and working with partially known hypergraphs (a far more realistic scenario in real-world hypergraphs such as social networks, where it is not possible to know the state of the entire network at any one given time).

This chapter has proven the applicability of the architecture-aware restreaming approach on a synthetic runtime benchmark. Once the scaling limitations have been

addressed, chapter 6 shows the suitability of *HyperPRAW* on real applications such as SNN simulations in large distributed systems.

To evaluate *HyperPRAW* this chapter has used a selection of hypergraphs from a known hypergraph dataset. Though these are sufficient to demonstrate the improvement brought by architecture-aware partitioning, it limits the analysis that can be made based on hypergraph parameters. Relevant parameters that may impact the performance of partitioning algorithms are vertex and hyperedge cardinality, average vertex and hyperedge degree distribution, number and size of hidden clusters and the size of the hypergraph. A hypergraph generator that can produce hypergraphs of specific profiles is therefore highly desirable. To date, there are no such parametric generators available. Chapter 5 proposes a simple and performant parametric hypergraph generator.

For simplicity, *HyperPRAW* does not model dynamic communication patterns or asymmetric communication patterns (where some hyperedges may communicate more than others). Both could be tackled by weighing hyperedges and consider the cost of partitioning accordingly, which is easily accommodated into *HyperPRAW* by weighing the cost of communications in the vertex assignment objective function with the hyperedge weight. This is an interesting area of future work that may impact performance in highly dynamic asymmetric parallel applications.

Chapter 5

A novel parametric hypergraph generator

5.1 Overview

In chapter 4 a collection of hypergraphs from hypergraph-related competitions is used to demonstrate the positive impact of architecture-awareness in performance of distributed applications that are modelled as hypergraphs. Although the results show significant communication speedup, the improvement varied across hypergraph models. In addition, the cardinality and vertex degree distribution of a hypergraph can have an impact both in the communication requirements of the modelled application and in the quality of partitioning of the graph. For instance, scale-free graphs (that exhibit power-law degree distribution in their vertex degree) have been shown to be better partitioned using hyperedge partitioning [189, 90] instead of vertex partitioning.

To better understand the impact of the properties of a hypergraph in graph tasks such as partitioning, it is important to sample the space of possible hypergraphs in a way that different categories of hypergraphs are represented. To achieve such a sampling, this chapter proposes a parametric hypergraph generator that can produce graphs that exhibit certain desired properties such as degree distribution, hyperedge cardinality and internal clustering. To the best of my knowledge there are no existing parametric hypergraph generators which go beyond basic metrics such as number of vertices and hyperedges and uniform degree distributions.

Thus, the contribution of this chapter (**C9**) is to **propose and evaluate a parametric hypergraph generator that allows researchers to produce graphs that exhibit custom complex properties**, including parameters such as: hyperedge and vertex count, hyperedge cardinality distribution, vertex degree distribution, number of clusters, density of the clusters. The resulting hypergraphs can be used in any benchmark application that uses hypergraphs as models, such as Boolean Satisfiability, circuit design, and sparse matrix algebra. Spiking Neural Networks, as other biologically-plausible networks, have been shown to display internal clustering [39, 170] and very high connectivity (10^2 to 10^4 synapses per neuron). A hypergraph generator that produces SNN-like graphs can be used to evaluate downstream tasks such as workload allocation and communication optimisation.

5.2 Justification

When developing algorithms to optimise hypergraph processing and partitioning it is important to evaluate them with hypergraphs that exhibit a wide range of internal features to analyse limitations and strengths and estimate how they fare when facing real world hypergraphs. In the case of hypergraphs used to model distributed applications (such as the ones presented in chapters 3, and 4) the nature of the graphs will vary depending on the nature of the application and its patterns of communication. Different types of hypergraphs categorised by different features are needed to simulate the range of applications they model and to understand the effectiveness of the partitioning approach. For instance, take the average cardinality, or number of vertices included on a hyperedge; hypergraphs with high average cardinality may represent applications with highly connected clusters of elements in which a message needs to be shared amongst a group of elements. Thus, research on partitioning hypergraphs for workload allocation requires considering graphs with different cluster density. Therefore, a procedural approach to generate such different graphs would be beneficial for researchers.

There are a number of parametric hyperedge generators in the literature, but they are limited in the features they can produce: [190, 220] cannot produce power law distribution of cardinality or vertex degree and unable to generate internal clusters; *SageMath*[1] has limited scalability and it is applicable only to small graphs. All reviewed generators are limited to only a few key parameters such as number of vertices and hyperedges.

Many real-world graphs, including biologically plausible networks, exhibit power-law distributions [142, 70, 39], hence it is important to generate hypergraphs that exhibit this. Power-law graphs are important as they are prominent in nature [170] and can be relevant to the way applications communicate (with hubs) that are hard to replicate with uniform graphs.

5.3 Approach

From the narrow case of graph generation (a specific hypergraph case where the hyperedge cardinality is equal to 2, i.e. the number of vertices joint by an edge is 2), there are two generators that inspired the proposed approach: [54] where graphs are defined by number of vertices and hyperedges, inter-cluster and intra-cluster edge probabilities; and [53], which allows the sampling of vertex degrees (number of edges per vertex) from a power-law distribution.

The proposed approach is a generative hypergraph process [190] in which hyperedges are created and added to the hypergraph one at a time, sampling vertices from a collection. The collection of vertices is sampled from a population based on the hidden partition model used in FENNEL [212] (but now expanded to accommodate for hyperedges), where vertices are defined by number of inter-cluster and intra-cluster connectivity probabilities.

We allow the size of the next hyperedge to be sampled from a power-law distribution. The vertices to generate each hyperedge are sampled from a separate

[1] `http://doc.sagemath.org/html/en/reference/graphs/sage/graphs/hypergraph_generators.html`

power-law distribution. This sampling distribution results in scale-free or power-law graphs. If a uniform distribution is preferred, a uniform sampling can be chosen instead.

The main parameters input for the generator, for a hypergraph $G = (V, E)$ where V is the set of vertices and E is the set of hyperedges are:

- Total number of vertices ($|V|$)
- Total set of hyperedges ($|E|$)
- Number of internal (hidden) clusters (k). A cluster is a subpopulation of vertices that exhibits higher intraconnectivity amongst member vertices.
- Cluster density: probability of a vertex to belong to a particular hidden cluster. $D = d_1, d_2, ..., d_k$ where $\sum_{n=1}^{k} d_n = 1$
- Probability of intra cluster connectivity (p): for a vertex belonging to a cluster, its probability to connect to a vertex belonging to a different cluster.
- Distribution of hyperedge cardinality $d_h(x) = c \times x^{-\gamma_h}$, where x is the size of the hyperedge, c is a normalisation constant and $d_h(x)$ is the power-law distribution, i.e. the probability function for each value of x. The distrubution is controlled by a γ_h parameter, which determines the skeweness of the power-law distribution to choose hyperedge sizes for any hyperedge. With $\gamma_h = 0$ the distribution is uniform. High values of γ_h result in highly skewed power-law distributions.
- Minimum (c_{min}) and maximum (c_{max}) hyperedge cardinality, or number of vertices per hyperedge.
- Distribution of vertex sampling $d_v(x) = c \times x^{-\gamma_v}$ where x is the vertex index to be selected when building the hyperedge, c is a normalisation constant and $d_v(x)$ is the power-law distribution, i.e. the probability function for each value of x. Like d_h, the distribution is controlled by γ_v.

The code for the current algorithm can be found at `https://github.com/cfmusoles/hypergraph_generator`.

Algorithm 2 lists the complete hypergraph generation procedure. First, a collection of vertices is created and assigned to one or more hidden clusters, following the defined cluster density. Then, hyperedges are created sequentially until the total number of hyperedges is reached. For each hyperedge, the algorithm decides how many vertex slots it will contain (sampled from the hyperedge cardinality distribution) and which cluster or population to target (random uniform). For each slot, it assigns either a local vertex (vertex from the target cluster) or a remote vertex (from any other cluster) with a probability set by the probablility of intra cluster connectivity. Irrespective of the origin of the vertex, the vertex is drawn from its population with a random distribution defined by the vertex sampling distribution (either uniform random or power-law).

Once the required hyperedges have been created, if required, the generator creates hyperedges (with size capped to maximum hyperedge cardinality) with any remaining vertices that have not been yet assigned to any hyperedge. This can be used to ensure that all vertices are at least present once in the graph.

Algorithm 2: Hypergraph generation algorithm

Input : $|V|$ number of vertices; $|E|$ number of hyperedges; k number of clusters; D cluster density; p intra cluster probability; d_h and d_v distributions; c_{min} and c_{max} cardinality range.

Output : Hypergraph $G = (V, E)$

Initialise V: Random generation of $|V|$ vertices and assign to C_k cluster set with k drawn from D distribution.

for $i = 1$ *to* $|E|$ **do**
- Create h_i empty hyperedge
- Size of hyperedge $size_e$ sampled from $d_e(x)$
- Local cluster c sampled from 1 to k
- **for** $j = 1$ *to* $size_e$ **do**
  - $prob$ sampled from 0 to 1
  - **if** $prob < p$ **then**
    - Vertex $v \leftarrow C_c$ drawn using distribution d_v
    - Add v to h_i
  - **else**
    - $cluster$ sampled from 1 to k, where $cluster! = c$
    - Vertex $v \leftarrow C_{cluster}$ drawn using distribution d_v
    - Add v to h_i
  - Add v to W visited vertices
  - Add h_i to G

if $|W| > 0$ **then**
- **for** $i = 1$ *to* $|W|$ **do**
  - Create h empty hyperedge
  - **for** $i = 1$ *to* $min(c_{max}, |W|)$ **do**
    - Add $w[i]$ to h
    - Remove $w[i]$ from W
  - Add h to G

return G

5.4 Evaluation

To demonstrate the correctness of the hypergraph generator, a series of hypergraphs are generated using the procedure described in algorithm 2. Once generated, the hypergraphs are analysed to prove that they show the expected properties as defined by the input parameters. The influence of each input parameter is demonstrated:

- Number of clusters (k) in section 5.4.1
- Cluster density ($D = d_1, d_2, ..., d_k$) in section 5.4.2
- Cluster recoverability in section 5.4.3 to evidence that the vertices are being drawn from the hidden cluster populations under different values of intraconnectivity p.
- Hyperedge cardinality (c_{min} and c_{max}) in section 5.4.5
- Cardinality distribution ($d_h(x)$) in section 5.4.4
- Vertex sampling distribution ($d_v(x)$) in section 5.4.4

All properties in the generated hypergraphs are kept equal, unless otherwise stated for each evaluation:

- 100000 Vertices.
- 100000 Hyperedges.
- Uniform cluster density.
- Cluster intraconnectivity $p = 1.0$ (i.e. 0 inter connectivity).
- Hyperedge cardinality range 10, 100.
- Uniform vertex sampling distribution ($\gamma_v = 0.0$).
- Powerlaw hyperedge cardinality distribution ($\gamma_h = 1.8$).

5.4.1 Number of clusters

The number and density of clusters in a hypergraph have a strong influence in the success of partitioning algorithms; due to the high interconnectivity of vertices within a cluster, dividing a cluster into two partitions would incur in higher hyperedge cut (a common hypergraph partitioning quality metric) than splitting partitions at cluster boundaries. Therefore, when evaluating partitioning algorithms it is important to include hypergraphs with different hidden cluster properties such as the number of clusters and their density.

To demonstrate the proposed algorithm can generate a specific number of clusters, hypergraphs generated with various number of clusters are fed to a state-of-the-art hypergraph partitioner (Zoltan [31]). Uniform density (same number of vertices per cluster) and no inter-cluster connectivity is assumed for this evaluation. If the hypergraph contains the same number of clusters as partitions requested from the

Table 5.1: Cluster size results that show how the hypergraph generator does form the specified number of clusters. AR and AMI are only applicable when the number of populations match between compared clusters.

	Original clusters	**Partitions**	**Hyperedge cut**	**AR**	**AMI**
cluster_5	5	5	0.0	1.0	1.0
cluster_5	5	12	0.972	N/A	N/A
cluster_12	12	5	0.107	N/A	N/A
cluster_12	12	12	0.0	1.0	1.0

partitioner, the number of hyperedges cut must be zero —the partitioner will divide vertices at the cluster boundaries, since no inter-cluster connectivity exists. If the values are not equal, the hyperedges cut must be higher than zero —the partitioner will divide at least one cluster into two different partitions. Table 5.1 shows the results obtained for two generated hypergraphs with 5 and 12 clusters. The partitioner is asked to partition each graph to 5 and 12 partitions. As expected, when the partitioner partitions the hypergraph with the same number of clusters as parts, the resulting hyperedge cut is 0.

In addition to hyperedge cut, two standard cluster similarity metrics are used to compare the original clustering output by the hypergraph generator and the partitioning results from the partitioner: Adjusted Mutual Information (AMI) and Adjusted Random (AR) scores [2]. Both metrics have a range score between 0 (no similarity) and 1 (identical), by comparing the vertices that are assigned to the two sets of clusters. Table 5.1 shows that quality of partition is higher when the number of clusters match the number of partitions (1 AR and AMI). When the two values do not match, the hyperedge cut is non zero. This demonstrates that the hypergraph generator has produced graphs from the desired hidden cluster properties.

5.4.2 Cluster density

Cluster density (i.e. relative proportion of vertices in each cluster) is rarely uniform in real world hypergraphs. Thus it is important that a hypergraph generator is able to produce custom cluster densities. To demonstrate that the proposed algorithm can generate custom cluster densities, the expected density ratio input to the generator (ratios for each cluster, where all ratios added up to 1) is compared to the actual vertex proportion (percentage of vertex in a cluster over the total). Since the generator knows what vertices belong to which clusters, this can be done with a simple look-up.

Figure 5.1 shows the vertex ratio and cluster density for two generated hypergraphs, for all clusters. On the left, with uniform cluster density (same expected density for all partitions); on the right, with custom cluster density (different expected densities). The results show that both vertex ratio and cluster density match, demonstrating that the generator algorithm can produce custom cluster density.

[2]The *scikit-learn* implementation of both algorithms is used in this work: `https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics`

(a) (b)

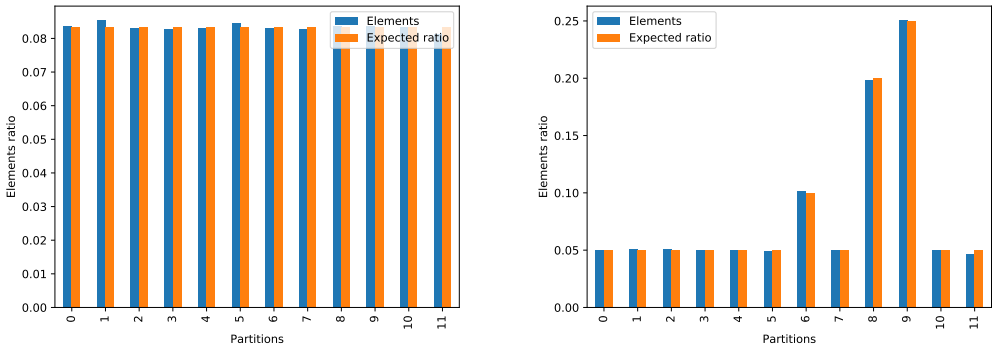


Figure 5.1: Cluster density measured in the ratio of vertices belonging to each cluster (orange) vs expected cluster density (blue) input to the generator algorithm. **A**: hypergraph with *uniform* cluster probabilities. **B**: hypergraph with *custom* cluster density ([0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.1, 0.05, 0.20, 0.25, 0.05, 0.05]).

5.4.3 Cluster recoverability based on intraconnectivity

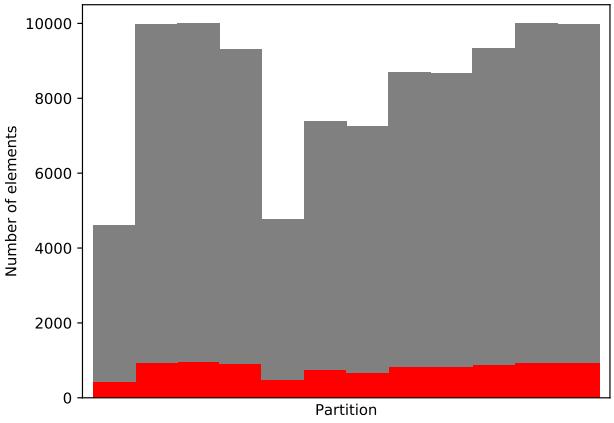
In section 5.4.1 the generator used cluster intraconnectivity $p = 1$, which results in independent, non overlapping clusters. In real world hypergraphs this is highly unlikely and therefore a desired generator should be able to produce clusters with some degree of inter-connectivity.

More inter-connected clusters (i.e. lower values of p) would result in harder to partition hypergraphs. A hypergraph that has intraconnectivity of 0.5 results in partitions that are dissimilar to the original clusters featured in the hypergraph, since the probability of a vertex to connect with a neighbour belonging to the same cluster is equal to the probability of connecting to an external cluster —this makes it hard for the partitioner to group vertices and effectively splits them randomly. To help measure this, a new metric is proposed: *Cluster Recoverability (CR)* factor per partition. The CR on a resulting partition is measured as the highest ratio of vertices belonging to the same original cluster. For example, if a partition contains 75% of vertices belonging to cluster A and the rest belonging to clusters B (10%) and C (15%), then the CR for that partition is 0.75. On a randomly connected hypergraph, the expected CR for each partition is $\frac{1}{P}$, where P is the number of partitions. The expected average CR increases with the probability of intraconnectivity p.

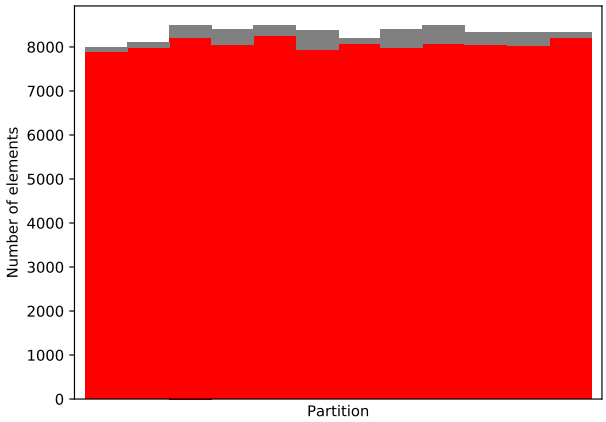
Figure 5.2 shows the CR for hypergraphs after being partitioned. As expected, when the vertices are randomly connected ($p = 0.5$), the CR is low (approximately equal to $\frac{1}{12}$). As the probability of intraconnectivity increases, the CR is higher. For hypergraphs with $p = 1$ (total intraconnectivity), the CR is 1.0, indicating that the partitions and clusters are identical.

To complement the results shown by CR, table 5.2 shows quality partitioning metrics (hyperedge cut) and cluster similarity metrics (AMI and AR) for the same hypergraphs. The hypergraph with $p = 0.5$, or random connectivity, shows poor partitioning quality (all hyperedges are cut) and similarity (near 0 AR and AMI). Greater values of intraconnectivity show, as expected, better quality and similarity scores. Note that even though the hyperedge cut may be high (as is the case with graph *p_conn_0_85*), as long as intraconnectivity is high enough, clusters can be

(a)



(b)



(c)

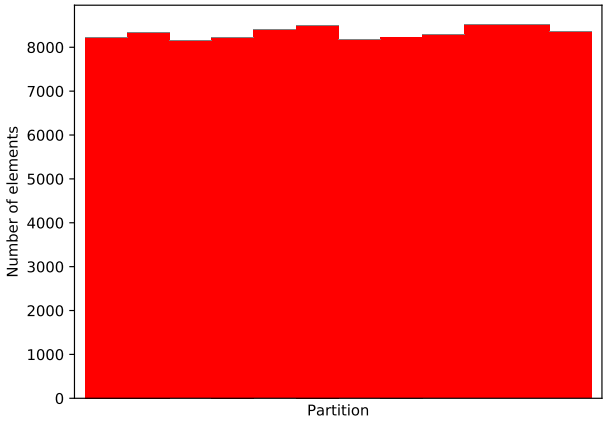


Figure 5.2: Cluster Recoverability (CR) for each partition on hypergraphs featuring different levels of intraconnectivity. **A**: intraconnectivity 0.5, **B**: intraconnectivity 0.85, **C**: intraconnectivity 1.0.

Table 5.2: Probability of intraconnectivity determines partitionability of graphs

	p_conn	**Hyperedge cut**	**AR**	**AMI**
p_conn_0_5	0.5	1.0	0.001	0.001
p_conn_0_85	0.85	0.848	0.931	0.915
p_conn_1_0	1.0	0.0	0.997	0.996

recovered efficiently (5.2B).

5.4.4 Cardinality and vertex degree distribution

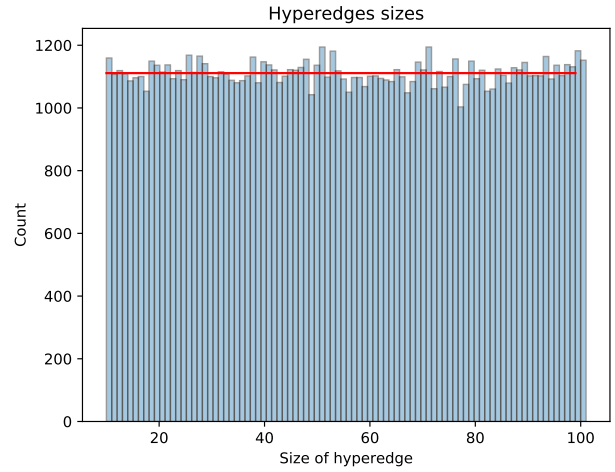
Many real world graphs and hypergraphs exhibit power-law distributions [142, 70, 39], both in the distribution of hyperedge cardinality (size of hyperedges) and vertex degree (number of hyperedges a vertex belongs to). This is an important feature of graphs that make few vertices highly inter-connected, and it has been shown to impact the type of partitioning techniques that are suitable [189, 90]. Therefore it is important to include it in a hypergraph generator.

To demonstrate the degree of distribution of hypergraphs generated with and without power-law distributions, hyperedge cardinality histograms are shown. Figure 5.3 depicts the distribution of hyperedge cardinalities as the number of hyperedges with each possible cardinality value. Three hypergraphs are shown, where the cardinality was ranged but followed different $d_h()$ distributions: uniform (figure 5.3A), power-law with $\gamma_h = 1.0$ (figure 5.3B) and power-law with $\gamma_h = 2.0$ (figure 5.3C). As expected, the higher the γ_h value, the more skewed the cardinality distribution appears.

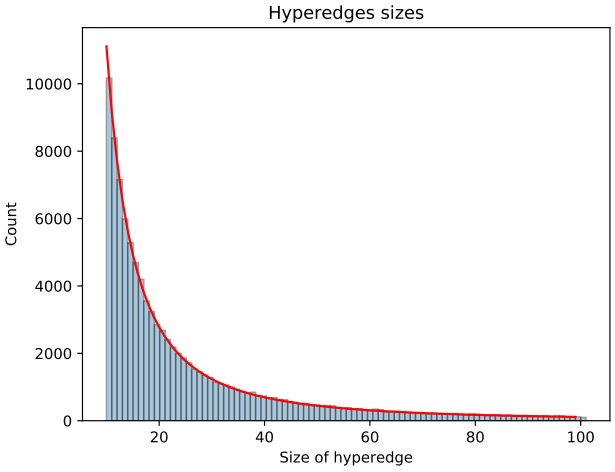
Similarly, vertex sampling histograms are used to show the impact of using power-law distributions to sample vertex from a population. For this, the following properties were used to generate hypergraps: 10,000 vertices, 10,000 hyperedges, 1 cluster, with a variable γ_v to evaluate the effect of the vertex sampling distribution. Figure 5.4 shows the distribution of vertex across the hyperedge, counting the number of occurrences of each vertex id. For simplicity, the underlying generator samples vertices from a single population (cluster) using either a uniform (figure 5.6A) or a powerlaw sampling distribution (figures 5.6B and 5.6C). The vertex sampling distribution matches the expected count of vertices (in red). For the uniform distribution, this is an average number of hyperedges containing each vertex. In the case of powerlaw distribution, the expected number of hyperedges containing a particular vertex (its vertex degree) is determined by a compound probability based on the powerlaw distribution. Each time a hyperedge draws a vertex from the population it uses the vertex sampling distribution (powerlaw), doing so as many times as the cardinality of the hyperedge. The figure shows that higher γ_v results in more skewed vertex sampling distributions as expected.

Figure 5.5 demonstrates the effect that hyperedge cardinality has over the vertex distribution. As more vertex are sampled to construct the hyperedge (higher cardinality), the probability of any vertex being present is a compound probability (multiple samples) in which each sample i governed by the powerlaw distribution selected. This results in higher probability verties being selected more often, the higher the cardinality. Figure 5.5A shows the vertex degree histogram for hyperedge cardinality equals 2, which closely resembles the corresponding powerlaw distribution

(a)



(b)



(c)

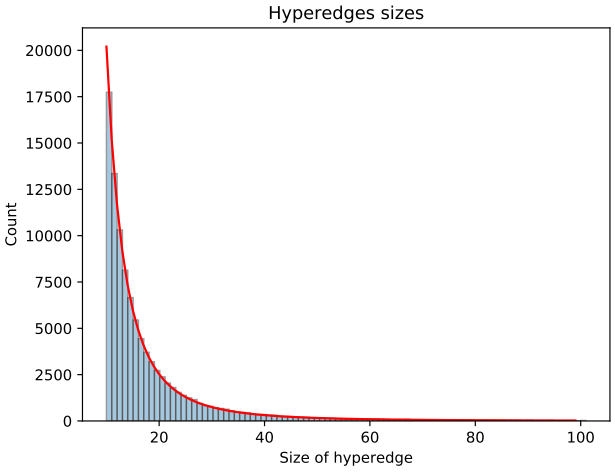
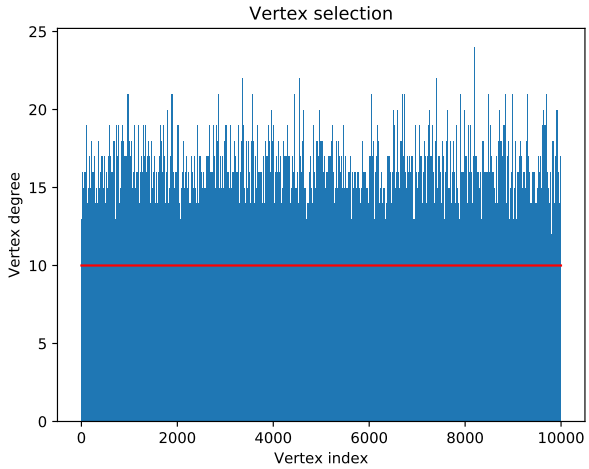
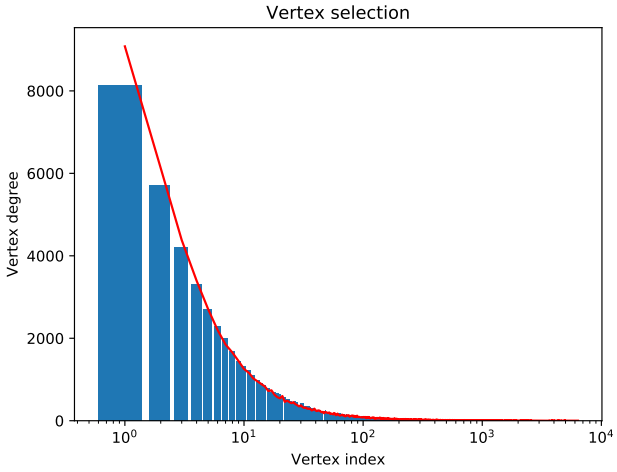


Figure 5.3: Hyperedge cardinality power-law distribution demonstrated by counting the size of hyperedges in hypergraphs generated under different values of γ_h of a power-law distribution. **A**: $\gamma_h = 0$ (equivalent to uniform distribution); **B**: $\gamma_h = 1$. **C**: $\gamma_h = 2$. Red line represents the expected distribution for the given γ_h.

(a)



(b)



(c)

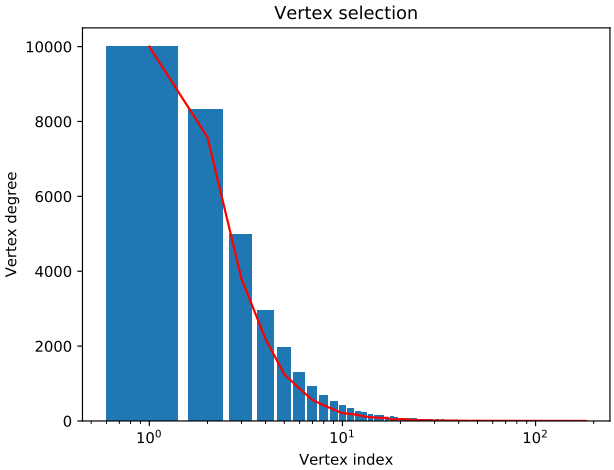


Figure 5.4: Vertex sampling power-law distribution demonstrated by counting the vertex degree in a population generated under different values of γ_v of a power-law distribution. **A**: $\gamma_v = 0$ (equivalent to uniform distribution); **B**: $\gamma_v = 0.2$. **C**: $\gamma_v = 1.5$.

for $\gamma_v = 0.5$ —since for each hyperedge, only two samples occur. In figure 5.5B, the hyperedge cardinality is 20, resulting in a much more skewed distribution. In both cases, the results matched the expected compound probability.

It is worth noting that when sampling multiple times, particularly in powerlaw distributions, the likelyhood of drawing the same vertex twice within the same hyperedge increases. In real life graphs, it is meaningless to have the same vertex twice; hence, the generator avoids adding a vertex when it is already present. The generative algorithm can choose what to do if a vertex is duplicated: skip it and move on to the next slot reducing the cardinality of the hyperedge (figures 5.5A and 5.5B) or resample from the population. Resampling is a sensible strategy to keep the hyperedge cardinality, but it affects the vertex degree distribution. Figure 5.5C shows how resampling increases the expected presence of vertices in the hypergraph due to resampling. Users must choose between accurate vertex sampling or ensuring hyperedge cardinality, which is a limitation of a generative approach.

Figure 5.5 demonstrates that our generative algorithm can sample vertices from uniform and powerlaw distributions. However this does not exactly result in vertex degree distribution that matches the sampling distribution. This is shown in figure 5.6, where a histogram of the frequency of vertex degrees across the hypergraph has a normal distribution. Although this is expected (due to the Central Limit Theorem in statistics), it may not be what the user has in mind when generating a hypergraph. If a vertex degree distribution needs to be exactly defined, the algorithm can be reinterpreted as a vertex generation process instead of a hyperedge one without further changes, by swapping vertex ids for hyperedge ids. The graph produced can be interpreted as folows: each element produced, instead of a hyperedge, is a vertex; and each id (or pin) within that element is a hyperedge id where the vertex is present. In this case, the user can accurately define the vertex degree distribution function and the hyperedge sampling distribution, but only approximate the hyperedge cardinality distribution. This is a limitation of a generative approach.

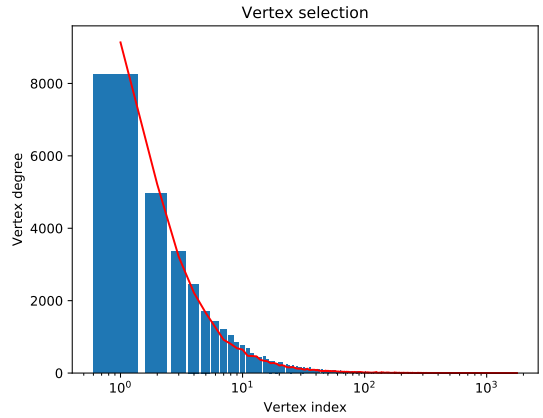
5.4.5 Size boundaries for hyperedge cardinality

The last parameter to evaluate is the hyperedge cardinality range, i.e. the maximum and minimum permissible hyperedge size. Figure 5.7 shows two hypergraphs generated with two sets of hyperedge cardinality ranges. Figure 5.7A presents the distribution of cardinality for the input range [2, 10], whereas figure 5.7B shows the distribution for range [50, 200], matching the input parameters passed to the generator. The plots demonstrate that the hypergraph generator can produce ranged hypergraph cardinality, and that it can maintain the appropriate power-law cardinality distribution that is within the input range as expected.

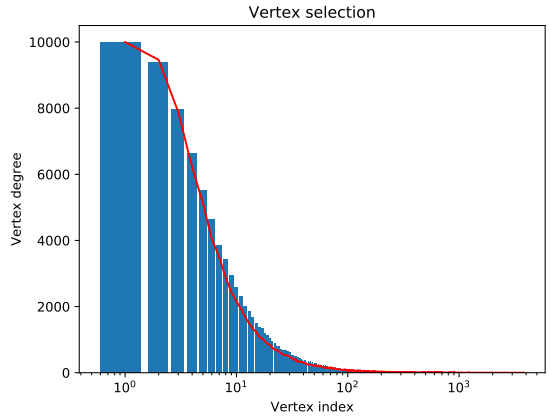
5.5 Conclusion

Tasks that use hypergraphs as models (boolean satisfiability, circuit design, sparse matrix multiplication, partitioning, communication in distributed applications) require a wide range of hypergraphs to evaluate and analyse their effectiveness. Up to now, algorithms have been tested with limited collection of handcrafted or handpicked hypergraphs. This makes it difficult to study algorithms against correlated features, and in that case it is difficult to find a hypergraph that matches

(a)



(b)



(c)

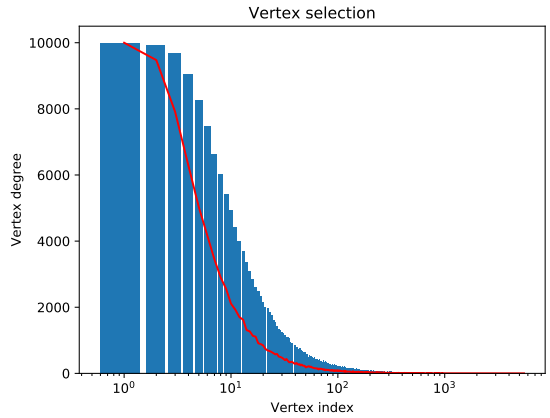


Figure 5.5: Effect that hyperedge cardinality (number of vertices) has over the vertex sampling distribution. All graphs are generated using $\gamma_v = 0.5$. Different hyperedge cardinalities used in **A** (5) and **B** (20), showing how sampling more often within the same hyperedge skews the powerlaw distribution. **C** shows the effect of resampling when the same vertex is drawn twice: when reselecting, more vertices are used (avoiding duplicates) which results in increased vertex degree (higher than the expected value).

(a)

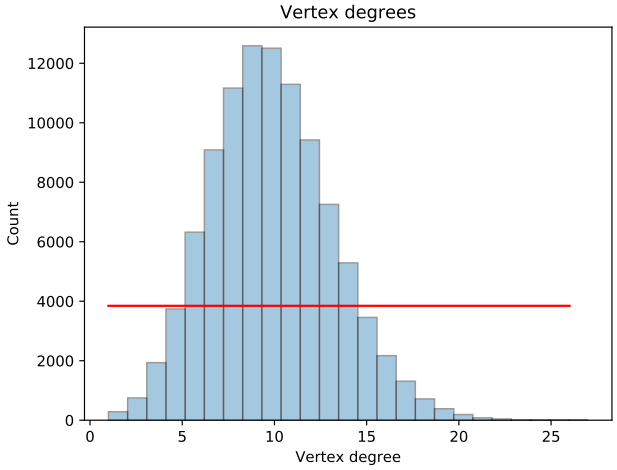


Figure 5.6: Vertex degree power-law distribution, counting the vertex degree in hypergraphs generated with a uniform random vertex sampling distribution. In red, expected values for a uniform random vertex degree distribution, which highlights the difference between vertex sampling and vertex degree distributions.

(a) (b)

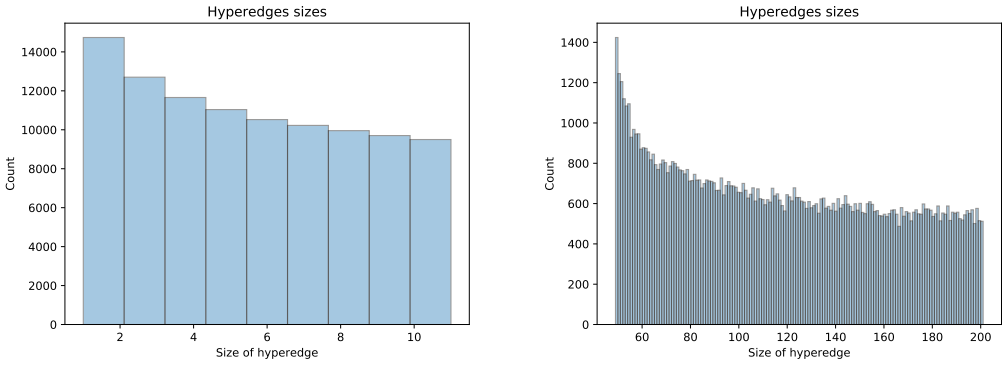


Figure 5.7: Hyperedge cardinality boundaries. **A**: [2, 10]; **B**: [50, 200]. Both hypergraphs are generated using a power-law distribution for hyperedge cardinality.

those.

This chapter **has demonstrated an effective parametric algorithm to generate custom hypergraphs that exhibit desired properties**, listed below:

- Total number of vertices.
- Total set of hyperedges.
- Number of internal (hidden) clusters.
- Cluster density.
- Probability of intra cluster connectivity.
- Distribution of hyperedge cardinality.
- Minimum and maximum hyperedge cardinality.
- Distribution of vertex sampling.

In the context of SNN simulation, the hypergraph generator presented in this chapter produces SNN-like graphs that can be used to evaluate downstream tasks such as workload allocation and communication optimisation.

The proposed hypergraph generator can aid research in the wider partitioning and processing algorithms that use hypergraphs as models. In chapter 6, the hypergraph generator proposed here is used to help characterise a novel parallel streaming hypergraph partitioning algorithm to aid workload allocation in distributed applications.

5.5.1 Limitations

As discussed in the chapter, due to the generative nature of our approach (growing the hypergraph one hyperedge or vertex at a time), there are currently some limitations to what the user can control:

- Vertex sampling distribution versus hyperedge cardinality guarantee: since vertices are sampled one by one to generate the hyperedge, the algorithm must choose what to do when the same vertex is drawn more than once; resampling results in a deviation from the expected vertex sampling distribution, whereas skipping the vertex reduces the expected hyperedge cardinality. One must choose between the two based on user needs.
- Some vertices may not be selected with a vertex sampling distribution, which may result in smaller graphs than expected. This can be corrected with adding new hyperedges that contained missing vertices, but this would skew both the hyperedge cardinality and the vertex distribution.
- The algorithm can describe precisely hyperedge size distribution or vertex degree distribution by reinterpreting the output graph (each element as a hyperedge or vertex), but not both.

Chapter 6

HyperPRAW: Architecture-Aware Parallel streaming hypergraph partitioning to model large scale distributed applications

6.1 Overview

Chapter 3 and chapter 4 show that SNN simulations can be further parallelised (strong scaling, using more computing nodes) with the use of architecture-aware hypergraph partitioning. To achieve the field's ambition of brain-scale simulations [150], (weak) scaling to large graphs is required. Whilst challenging, whole brain simulations are not alone in the requirement for data processing of large scale graphs and hypergraps. In the era of social media, such large graphs and hypergraphs are increasingly relevant in many scientific applications [122] and analytics. Similarly to neural networks, these vast graphs are too big for a single machine to process, both in terms of performance and memory requirements. State-of-the-art graph partitioning uses multilevel partitioning algorithms (based on [103]). The literature review in Chapter 2 concluded that these approaches have been shown to struggle to scale [49, 125] with large graphs on distributed memory. Streaming partitioning, in which elements (vertices or edges) are presented one at a time, is an optimised performance alternative to multilevel partitioning [46, 47, 212, 153].

The earlier work in chapter 4 showed that partitioning the hypergraph that models a distributed application communication patterns employing network bandwidth data leads to significant runtime speedup (up to 14x) compared to architecture-agnostic partitioner. However, as a sequential, restreaming algorithm, it operates under strong performance constraints, which result in long partitioning times (hours for large hypergraphs). This is worsen by the need to do several passes to refine the partitioning quality. The performance limitations of a sequential implementation reduce its applicability and appeal on large scale hypergraps.

This chapter explores ways of mitigating the above limitations and proposes a parallel streaming implementation for architecture-aware hypergraph partitioning (HyperPRAW). The **research question** is formulated as: **can an architecture-aware streaming hypergraph partitioning retain performance whilst operating under scalable memory and processing time requirements?**

The process of parallelising a streaming partitioning approach requires some decentralisation of the data leading to distributed processing with an imperfect view of the entire graph. This inevitably makes parallel partitioning worse (in terms of partitioning performance) than an un-scalable approach with overvoew of the entire graph. Therefore, a key factor is to balance loss of partitioning quality (e.g. each stream operates with local information only) and performance boost (e.g. faster partitioning time and reduced memory requirements). The results show that HyperPRAW continues to improve synthetic simulation runtime (up to 5.2x speedup over sequential hypergraph partitioner and up to 4.3x speedup over global partitioner), whilst keeping a good, scalable partitioning performance (with comparable partitioning times to a global partitioner, and significantly less memory requirements). Importantly, HyperPRAW partitioning performance allows it to scale to larger, partially unknown hypergraphs that global partitioners cannot.

This work is relevant for large scale distributed applications in which the communication between computing nodes may be limiting performance and scalability. It can help SNN simulations to scale to meet the computational demands of brain-size simulations [73], an ambition of the field. Examples of other domains in which this work is applicable are: general irregular distributed applications [197] such as distributed n-body problem [2] and scientific simulations ([215, 214, 231]; iterative solvers for non symmetric linear systems [213] and sparse matrix multiplication [44, 18, 19, 5].

6.2 Contributions

The chapter contributions to the overall thesis are:

- **C10**: Describe a novel architecture-aware streaming hypergraph partitioning algorithm that can run in distributed streams efficiently without significant loss in quality.
- **C11**: Demonstrate the impact of workload balance, architecture-awareness and staggered streaming has on the quality and performance of the parallel streaming partitioning.
- **C12**: Faster runtime on modelled synthetic distributed applications using parallel streaming hypergraph partitioning by up to 5.2x over sequential streaming partitioners and up to 4.3x over state-of-the-art global partitioners.
- **C13**: Demonstrate scalable parallel streaming partitioning with reduced memory footprint compared to a global state-of-the-art partitioning algorithm that allows it to tackle larger hypergraphs.

The remainder of the chapter is organised as follows: section 6.3 discusses how to parallelise a streaming hypergraph partitioning algorithm and identifies workload balance and staggered streaming as key performance factors. This defines the base approach that is further improved in subsequent sections. Section 6.4 proposes a way to incorporate architecture awareness (network bandwidth) into the algorithm and compares the runtime on a synthetic simulation benchmark to that of a global parallel partitioner —Zoltan [62]. Finally, section 6.5 highlights the reduced memory requirements of the parallel streaming algorithm compared to a global partitioner.

6.3 Parallel streams of elements

Sequential streaming partitioning is time consuming and as hypergraphs get larger, time and memory requirements increase beyond the capabilities of a single computing node. To speed up the process, and to help reduce the memory requirements, the partitioning can be approached with parallel streams. For graph partitioning, Battaglino et al. [21] demonstrate that parallel streaming with minimal quality loss is possible by creating one stream per partition and periodically synchronising workload and partition assignments. The loss of quality can be mitigated by doing several passes (restreaming), until an adequate balance between workload balance and partition quality is reached. This is the approach we took in chapter 4.

Restreaming partitioning assumes the graph is static and known ahead of time. Those limitations are too restrictive for applications such as large social networks, in which graphs change rapidly. This chapter proposes a single pass streaming partitioning algorithm that extends to hypergraphs that works in parallel via multiple streams. This tackles the following limitations: reduced memory requirements by working on independent parts of the hypergraph in parallel; can handle partially unknown hypergraphs (single pass); and it is applicable to hypergraphs as well as graphs.

Streaming partitioning allocates a single element of the graph at any given time, taking only partial information —in contrast with complete information on global approaches. For vertex-centric partitioning, each element is a vertex and all the hyperedges in which it is present. Alternatively, in edge-centric partitioning, an element is a hyperedge and all the vertices contained in it. As discussed in section 2.8.1, edge-centric partitioning has been shown to be superior to vertex partitioning in power-law graphs [189, 90], graph processing [122] and distributed database entity placement [125, 227].

This work uses the following terminology when referring to streaming partitioning: an *element* is the unit of allocation for the partitioner. Each *stream* receives an element (either a vertex or a hyperedge) at a time, and makes a partitioning allocation based only on the information available at the time —i.e. previous allocations and the element itself. Multiple streams work on non-overlapping subsets of the hypergraph to divide the partitioning workload. The *allocation* decision consists of assigning the current element to one of the available partitions. Elements contain a list of *pins* (ids) —vertices contained in a given hyperedge, or a hyperedges a given vertex belongs to.

The algorithm proposed in this chapter can be applied as both vertex- and edge-centric by simply modifying the input stream of elements. If each element is a list of hyperedges that a vertex belongs to, the algorithm behaves as a vertex-centric; alternatively, if an element contains a list of vertices a hyperedge contains, it acts as edge-centric. This dual nature provides flexibility as the hypergraph can be partitioned based on the application domain that is going to use it.

6.3.1 Base approach: single pass multistreaming partitioning

In order to create a parallel implementation of an architecture-aware streaming partitioner, each feature of the algorithm is evaluated individually to understand its

impact to the overall result. The base approach consists of adding parallelism to an architecture-agnostic streaming hypergraph partitioning.

The proposed base approach is a parallel extension of [6]. It is a greedy, single pass approach in which elements are received one at a time and once allocated they are not reconsidered. In the base approach, the allocation value function is a pin overlap count. Informally, the more pins that are already present in a given partition, the higher the score for that partition candidate. The element is assigned to the partition with higher score. In case of equal score, the element is assigned to the least full partition.

The algorithm is formally described in Algorithm 3, where the $Overlap_k(N_e)$ function for partition k, N_e list of pins p belonging to element e is defined as:

$$Overlap_k(N_e) = \sum_{p \in N_e} \begin{cases} 1, & \text{if } p \in P_k \\ 0, & \text{otherwise} \end{cases} \tag{6.1}$$

The function $Overlap_k$ only uses information that is available to streams, i.e. past assignment information, stored as $seenPins_{e,k}$ for element e and partition k. After each element allocation, streams share the element allocation and the list of pins to update $seenPins_{e,k}$. In the current implementation, this synchronisation is done with all to all MPI communication after each element allocation.

Instead of having a single stream of elements, our approach allows for multiple streams. The number of streams does not have to be the same as the number of targeted partitions. Each substream performs the allocation value function independently, which speeds up the allocation process.

The load imbalance of a partitioning allocation is measured as the largest partition over the average partition size, where size is given by the element count. To achieve non-trivial partitioning allocations [1], load imbalance must be kept under a tolerance value. To maintain load balance, partitions that are full (element count reaches max allowed) are not considered for future assignments. The parallel algorithm is guaranteed to reach load imbalance tolerance if in the last round of streams assigning elements to partitions, an allocation cannot be moved from a position of acceptable imbalance to one outside the imbalance tolerance. The worst case scenario is when all streams assign the last element to the same partition. To ensure that this does not exit imbalance tolerance, the maximum cardinality of an element (number of pins, or workload associated with it) times the number of streams must be lower than the total workload times the imbalance tolerance:

$$W \times i > W + C_{max} \times s \tag{6.2}$$

where W is the expected workload for a stream (total workload over the number of partitions or streams), i is the imbalance tolerance (1 being perfect balance) and s is the number of streams. By definition, the total workload W is the total number of elements times the average cardinality of elements: $W = |E| \times \frac{|pins| \times C_{avg}}{p}$, where p is the number of partitions. Substituting and solving for s, imbalance tolerance is guaranteed provided the hypergraph satisfies:

[1] A trivial allocation with no edgecut is achieved by assigning all elements to the same partition. Although this results in zero edgecut (perfect partitioning quality), it is of little value in the context of workload distribution as the entire application works now sequentially.

$$s < \frac{|E| \times C_{avg} \times (i-1)}{C_{max} \times p} \tag{6.3}$$

In practice, if a hard guarantee is not required, C_{max} can be swapped for C_{avg}. To simplify, if $p = s$, solving for the number of streams s gives the soft limit:

$$s < \sqrt{(|E| \times i - |E|)}) \tag{6.4}$$

Algorithm 3: Parallel HyperPRAW: implementation of streaming hypergraph partitioning

Input : s number of parallel streams; p (number of partitions); ϵ (maximum imbalance tolerance)
Output : P_k, for $k = 1, ..., p$, where P_k is the subset of E that is allocated to partition k
Data : $H = (V, E)$, where V is a set of vertices and E is a set of hyperedges for hypergraph H.

Round robin population of streams $S_x(E, N)$ for s streams of E elements, and N pins; N_e denotes the pins belonging to element e.
Set $W(k) = 0$ for each partition k
Set $seenPins_i = \emptyset$ and $P_i = \emptyset$ for $i = 1$ to p
for s *in* S **do**
    $maxWorkload = \frac{|E|}{|N|} * \epsilon$
    for $e = 1$ *to* E **do**
        $candidateParts \leftarrow$ select all $k = 1, ..., p$ if $W(k) < maxWorkload$
        $j \leftarrow \arg\max_{k \in candidateParts} = Overlap_k(N_e)$
        Synchronise all s: store* (N_e, j)
        Add e to set P_j
        Add pin to $seenPins_{pin,j}$ for all $pin \in N_e$
        $W(j) += 1$
return P

6.3.2 Evaluation

To evaluate the impact of parallelising a streaming partitioner, scaling experiments with increasing number of streams are performed. 1, 4, 16 and 64 streams alternative are evaluated on a fixed task of partitioning the synthetic hypergraphs in 96 parts. The aim is to show partitioning quality degradation and partitioning speedup as a function of the number of streams. As a baseline, a sequential version (single stream) is included. This baseline approach is a custom implementation of the sequential streaming hypergraph vertex partitioner proposed by [6].

Using the parametric hypergraph generator proposed in chapter 5, a diverse collection of synthetic hypergraphs is generated. These hypergraphs are used in the scaling experiments to understand how the parallel streaming deals with different types of hypergraphs. The hypergraph parameters are chosen to sample a wide range of hypergraphs features in terms of hyperedge cardinality density, size of the hypergraph, vertex degree distribution and number of internal clusters. Using a

collection of synthetic graphs for the evaluation avoids potential biases that specific SNN models may have. Here is a summary that describe each feature:

- Hypergraph sizes: Small (50k hyperedges), Large (500k hyperedges), Huge ($>$ 800k hyperedges).
- Hyperedge cardinality range: Dense (den, 50 to 200), Sparse (sp, 4 to 20).
- Vertex degree distribution: Uniform (uni, uniform distribution), Powerlaw (pw, power-law distribution, with $gamma = 1$) —see section 5.4.4 for more details.
- Number of internal populations: 96 vertex clusters / populations (c96). 48 clusters (c48). 192 clusters (c192); where a population is a region of the graph with high intra-connectivity between vertices.

In all cases, other parameters are set equal (uniform cluster density, intra-connectivity of 0.99, resulting in clustered populations of vertices with some inter-cluster connectivity, which is representative of biologically plausible networks [170, 39]). For a detailed description of each hypergraph parameter, see chapter 5. Table 6.1 lists the most important characteristics of all synthetic hypergraphs. All hypergraphs are represented by very sparse matrices (where rows are vertices and columns are hyperedges), with a Number of Non-Zero (NNZ) elements ratio between 3×10^{-5} and 3×10^{-4} (for example, a ratio of 0.01 would indicate that 1% of the elements on the matrix are non-zero). This indicates very localised connectivity between vertices, which makes them good candidates for partitioning, and a good model of real world distributed applications that can benefit from partitioning —such as sparse matrix multiplication [44, 18, 19, 5] and large scale scientific simulations ([73, 215, 214, 231]. The hypergraphs generated share properties with SNN models, such as highly sparse connectivity and presence of internal populations or clusters.

6.3.3 Results of parallel streaming

Figure 6.1 demonstrates that multiple streams reduce partitioning time across all tested hypergraphs. The speedup is linear with the number of streams, until the total time approaches 0, where the speedup gains are balanced against the overheads of running the algorithm. Even though more streams accelerate the processing time, the speedup factor is less than the stream multiplier in certain graphs (power-law), highlighting the synchronisation costs of running parallel streams.

Streams working in parallel have to make allocation decisions based on local information only (information available since the last synchronisation event). To understand how multiple streams impact the quality of partitioning, Figure 6.2 shows the partitioning quality loss as streams are scaled up. Standard metrics of hyperedge cut (number of hyperedges that span across partitions) and Sum Of External Degree (SOED, or the number of participating partitions in all cut hyperedges) are used as a proxy for partitioning quality. For both quality metrics shown, SOED and hyperedge cut, the quality starts to degrade from 16 streams.

Figure 6.1 and 6.2 show that partitioning with multiple streams achieves faster processing time as expected, at the expense of quality degradation. However, it is unclear what is behind the quality loss. Section 6.3.4 investigates this by characterising the quality degradation and proposes a mitigation strategy.

Table 6.1: Synthetic hypergraphs generated for benchmarking

Name	V	H	Avg. H cardinality	V degree distrib.	Clusters	Total NNZ (ratio)
large pw sp 96	200,000	500,344	6.24	power-law	96	3,122,874 (3×10^{-5})
small pw den 96	200,000	50,018	58.45	power-law	96	2,923,497 (3×10^{-4})
large uni sp 96	200,000	500,000	6.24	uniform	96	3,118,515 (3.1×10^{-5})
small uni den 96	200,000	50,000	58.51	uniform	96	2,925,264 (2.9×10^{-4})
small uni sp 96	200,000	52,121	6.78	uniform	96	353,129 (3.4×10^{-5})
small uni sp 48	200,000	52,106	6.79	uniform	48	353,639 (3×10^{-5})
small uni den 192	200,000	50,000	58.37	uniform	192	2,918,637 (3×10^{-5})
huge uni den 96	1,000,000	1,700,000	16.33	uniform	96	27,767,042 (3×10^{-5})
huge uni pack 192	800,000	800,000	86.76	uniform	192	69,403,637 (3×10^{-5})

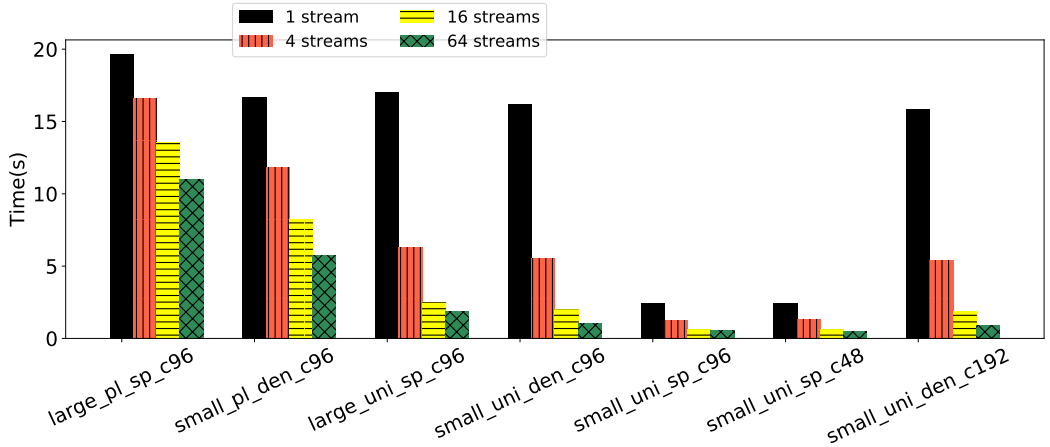
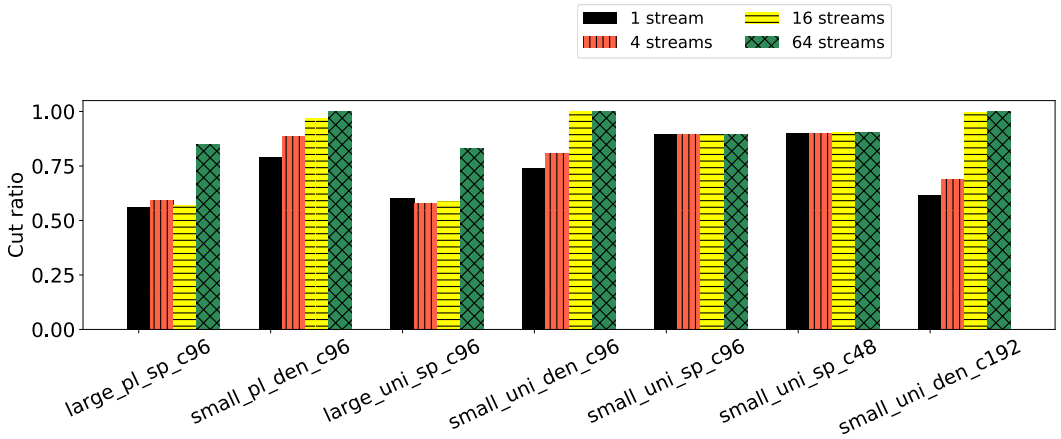


Figure 6.1: Partitioning time gains with increased parallelism (multiple streams) on a fixed task of partitioning hypergraphs in 96 parts. 1, 4, 16 and 64 streams are compared, showing reduction in partitioning time as the number of streams increases.

(a)



(b)

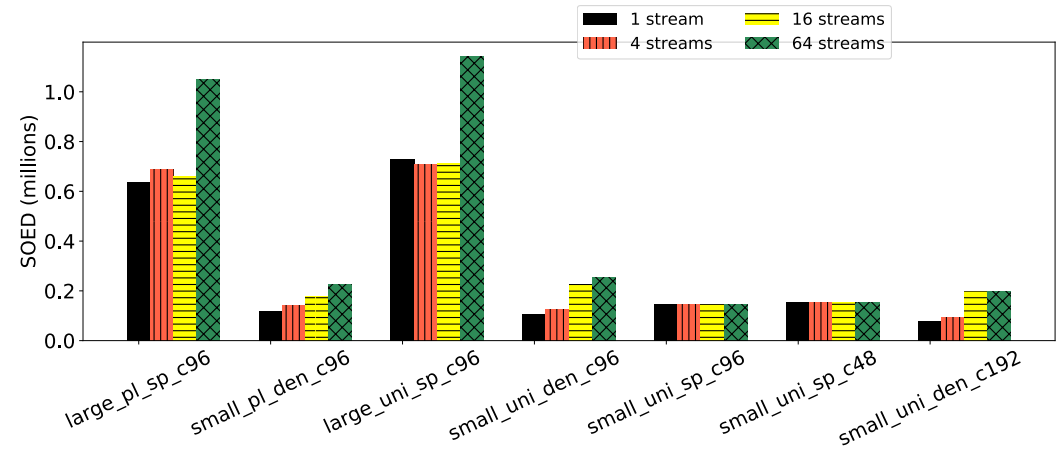


Figure 6.2: Partitioning quality loss with increased parallelism (multiple streams) on a fixed task of partitioning hypergraphs in 96 parts. 1, 4, 16 and 64 streams are compared. **Top**: Hyperedge cut ratio; **bottom**: Sum Of External Degrees (SOED). The quality starts to degrade from 16 streams upwards across most hypergraphs.

6.3.4 Staggered vs uniform streams

To demonstrate the cause of quality degradation as the number of streams increases, Figure 6.3 shows the cluster recoverability (CR) plots for scaling hypergraph partitions (only one hypergraph is shown here). Recall from section 5.4.3 that CR on each part on a partition is measured as the highest ratio of vertices belonging to the same original cluster (true cluster). A high average CR (highlighted in red) indicates the partitioning algorithm is able to recover the original clustering (which is known in these synthetic graphs). The top part of Figure 6.3 shows that CR is poor at 16 and 64 parallel streams.

Recall that the allocation value function in Algorithm 3 evaluates candidate partitions sequentially and chooses the candidate with highest score. If two candidates have the same score, the element is assigned to the partition with lower workload, or the first one seen if all is equal. Since all parallel streams follow the same procedure, initial partition candidates (with lower ids) have a higher chance of being used first. At large numbers of streams, this has a compound effect that does a poor job at keeping connected elements together, since it attempts to fill initial partitions first —overlap would be higher on those candidates, which encourages filling those partitions. Top Figure 6.3 shows how initial partitions at high stream counts have poor CR as a consequence.

To mitigate the compounding effect of loading initial partitions first, we propose a staggered start for candidate evaluation, in which each stream starts evaluating candidates on the n partition:

$$n = round(\frac{streamID}{s} \times |P|) \tag{6.5}$$

where $streamID$ is the unique ID of the stream, s is the number of streams and P is the set of partitions. Note that this does not affect the score for partition candidates, it just avoids all streams compounding allocations on the same partition candidate. Bottom Figure 6.3 shows the impact of staggered streaming, with no noticeable CR degradation at any scale.

A second effect of staggered streaming is that it distributes workload more evenly. This is because elements are assigned to initial partitions, henceforth making overlap more likely to happen on those partitions, assigning more elements to them. Then in the end, final processes are used —see 6.3C, where initial partitions tend to be full, whereas final ones tend to be under capacity. We call this the *tail-end effect*.

The staggered start also has a mitigation effect on the overall quality metrics degradation. Figure 6.4 show that a uniform start (non staggered) results in SOED degradation at high stream counts (16 and above) in three sample hypergraphs. However, a staggered start does not display degradation of SOED even at 64 streams.

6.3.5 Adding workload balance parameter

The streaming algorithm is guaranteed to keep a desired global workload balance by pre-computing the maximum load allowed for a single partition (imbalance tolerance times the average expected workload) and not assigning further elements to full partitions. However, this can still lead to large imbalances amongst individual partitions (*tail-end effect*). To avoid the effect, a workload parameter is incorporated

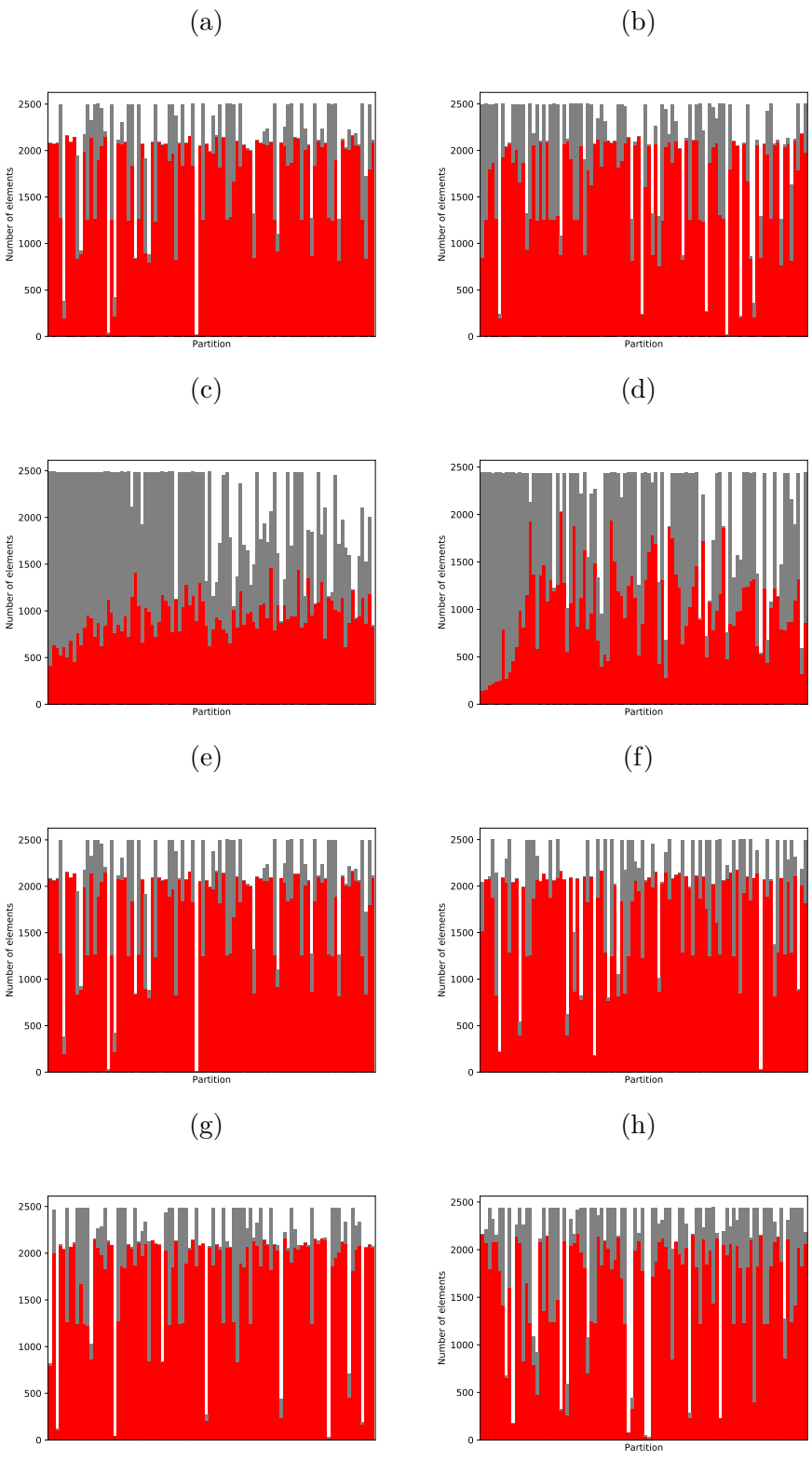


Figure 6.3: Cluster recoverability (CR) on non staggered and staggered streaming with increased parallelism. **Top**: non staggered streaming with 1 (**A**), 4 (**B**), 16 (**C**) and 64 (**D**) streams. **Bottom**: staggered streaming with 1 (**E**), 4 (**F**), 16 (**G**) and 64 (**H**) streams. Staggered approach maintains good CR at high streaming counts, whereas non staggered streaming CR is significantly affected from 16 streams upwards. Hypergraph used: small uniform dense 96.

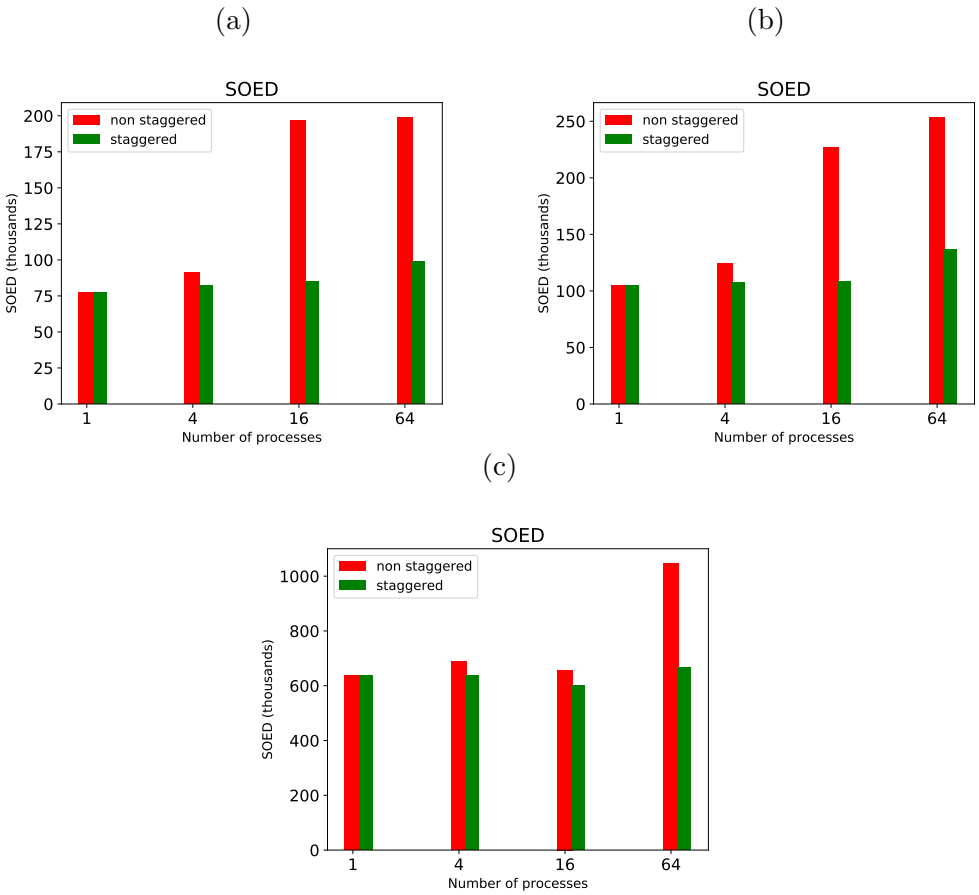


Figure 6.4: Comparing loss of partitioning quality in staggered (streams start evaluating partition candidates at different points) and non staggered (same start) streaming. Target partitioning: 96 parts. Hypergraphs: **A** small uniform dense 192, **B** small uniform dense 96, **C** large powerlaw sparse 96. Staggering the start of candidate partition across streams mitigates the quality degradation.

as part of the element allocation value function that encourages choosing candidate partitions with less workload.

From Algorithm 3, the candidate partition evaluation is modified to incorporate a parameter that is a function of the workload currently assigned to the partition, as shown in equation 6.6. λ is an extra input parameter that modulates the weight of workload imbalance differences in the total allocation value function.

$$j \leftarrow \underset{k \in candidateParts}{\arg\max} = Overlap_k(N_e) - W(k)^{\lambda} \tag{6.6}$$

λ is used as an exponent to produce a non-linear weighting of the workload imbalance. Since λ is in the [0-1] range, differences in workload matters more for relatively emptier partitions. This is advantageous to make the algorithm care less about workload differences between partition candidates when both of them are relatively full, paying more attention to the communication cost when partitions are about to be closed.

Figure 6.5 shows the effects of choosing different values for λ ($\lambda = 0$ representing no workload), with $\lambda = 0.5$ being the value that brings the best partitioning quality results (lower SOED). A too high λ results in partitions that have low quality (low SOED) as too much importance is given to workload, overpowering the overlap parameter —$\lambda = 1$.

Global workload imbalance only indicates that there will not be a single partition that has more than 1.2 times the expected average workload (if imbalance tolerance is set to 20%). However, this does not guarantee that all partitions will have similar workload, as many can be severely underutilised. Individual imbalance in parallel computations can lead to an increase of idle time between processes that need to synchronise information, as discussed in 3.4.10.3. Figure 6.6 shows the impact λ has on workload balance and cluster recoverability. On top, partitioning without workload balance parameter show great variances of individual process workload (the height of the bars). At the bottom, the same partitioning but using workload balance parameter; it is clear that more uniform individual workload balance is achieved, with small differences between elements assigned to any process. In terms of cluster recoverability, using a workload balance parameter has a small positive impact, with 5-15% higher Adjusted Random (AR) score in all stream counts —a common cluster comparison metric, discussed in 5.4.1.

6.4 Architecture aware parallel streaming

Chapter 4 demonstrated the impact of using information on network bandwidth communication to improve runtime performance of distributed applications by incorporating computing node peer to peer communication cost in streaming hypergraph partitioning. As a result, runtime simulation is sped up by mapping parallel communication application patterns to the underlying hardware network bandwidth patterns. This section explores modifications to the allocation value function in parallel streaming to incorporate network bandwidth communication with the same ambition of reducing simulation runtime on modelled distributed applications.

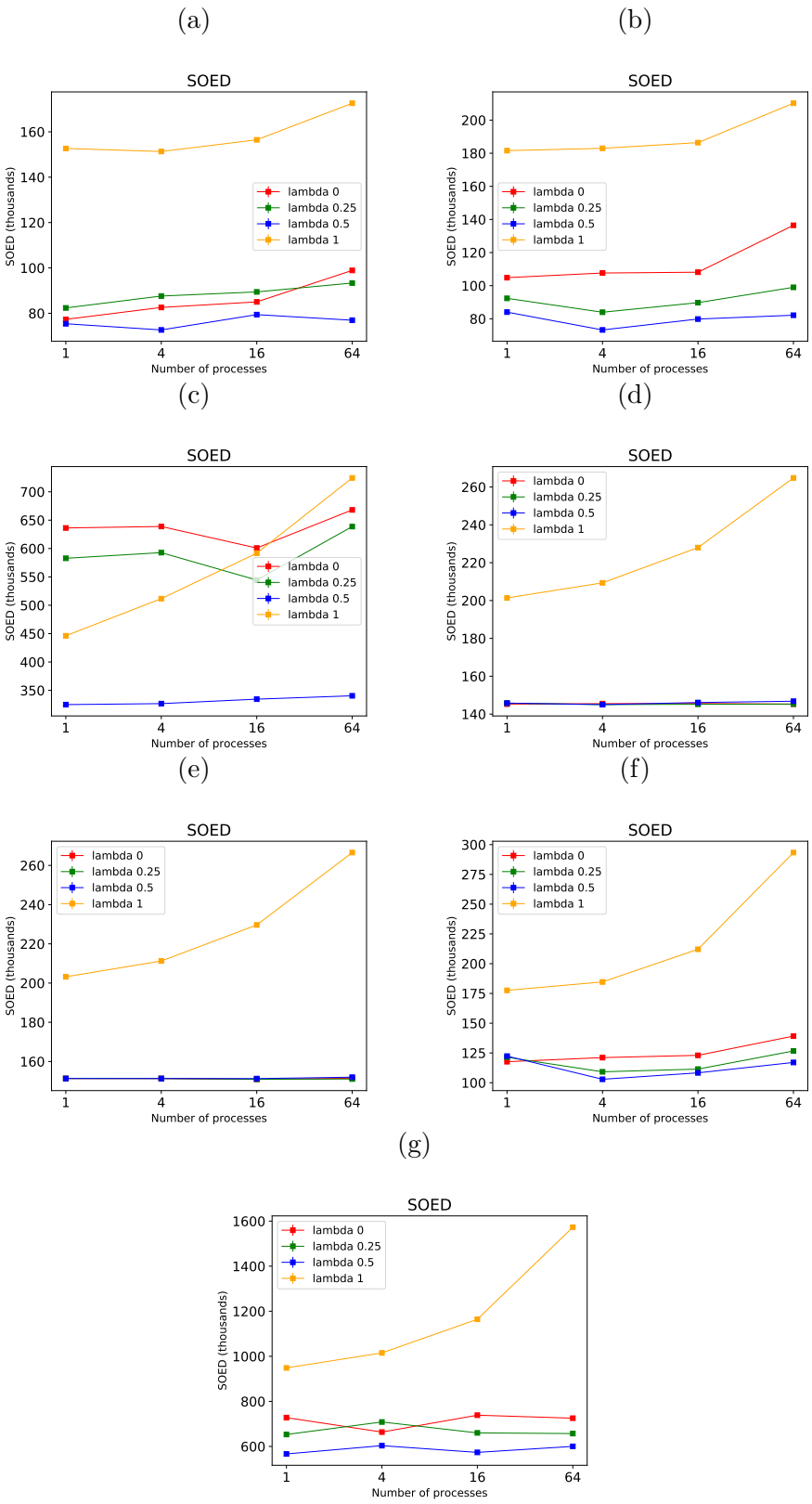


Figure 6.5: Partition quality loss in parallel streaming using an additional workload parameter in the element allocation value function. Target partitioning: 96 parts. Four different λ values are tested: 0 (no workload, **red**), 0.25 (**green**), 0.5 (**blue**) and 1 (**orange**). Hypergraphs used: **A** small uniform dense 192, **B** small uniform dense 96, **C** large powerlaw sparse 96, **D** small uniform sparse 96, **E** small uniform sparse 48, **F** small powerlaw dense 96, **G** large uniform sparse 96. $\lambda = 0.5$ results in negligible quality degradation and better (lower SOED) partitioning quality.

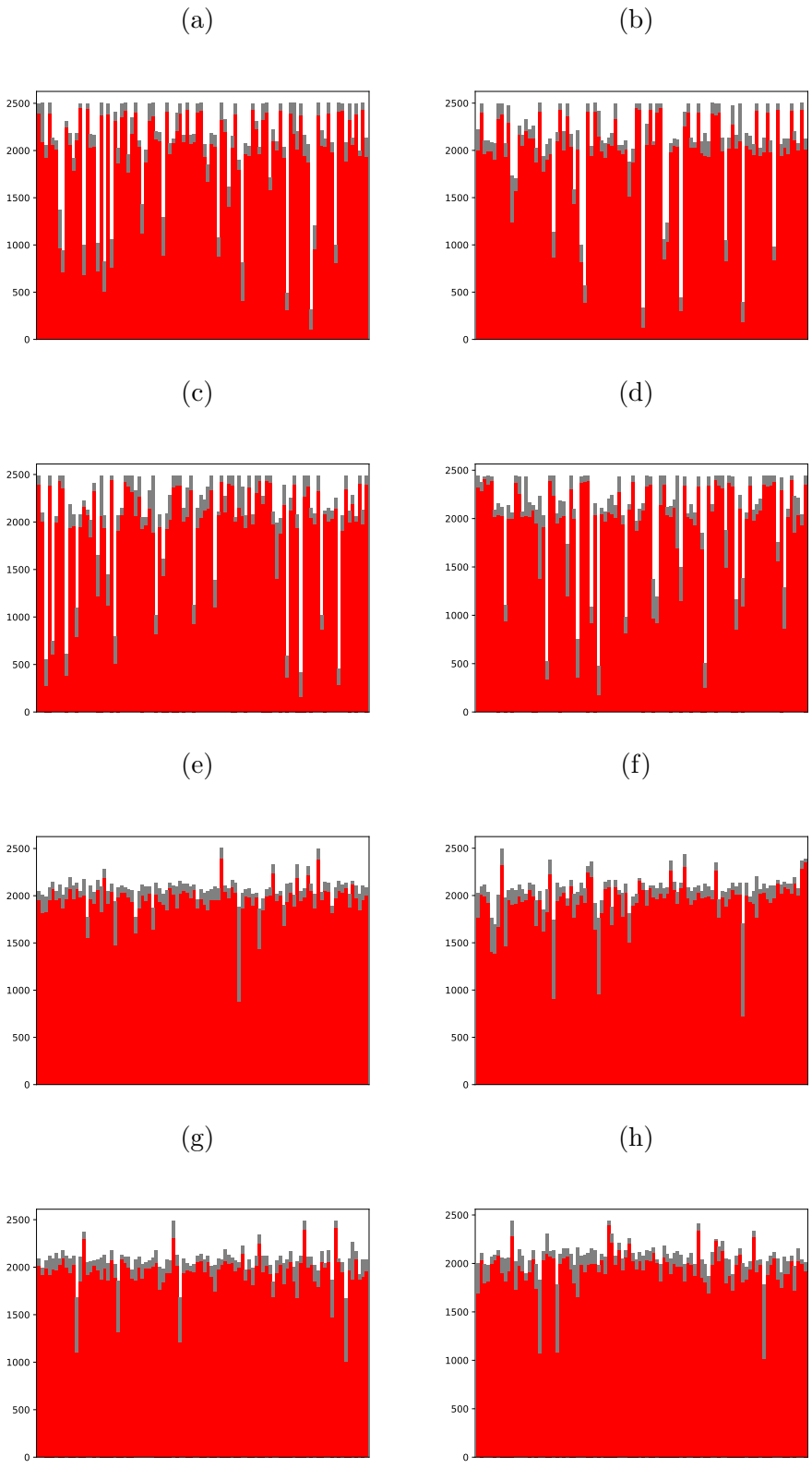


Figure 6.6: Impact of workload balancing parameter on cluster recoverability, Adjusted Random score and partition workload balance. **Top**: without workload balance parameter, increasing number of streams; 1 (**A**, 0.665 AR), 4 (**B**, 0.701 AR), 16 (**C**, 0.660 AR) and 64 (**D**, 0.667 AR). **Bottom**: with workload balance parameter, $\lambda = 0.5$, increasing number of streams; 1 (**E**, 0.765 AR), 4 (**F**, 0.740 AR), 16 (**G**, 0.762 AR) and 64 (**H**, 0.745 AR). Hypergraph used: large uniform sparse 96.

6.4.1 Incorporating communication cost

To incorporate communication cost in the allocation value function, first we introduce the concept of *pin replica*. A pin replica is each partition in which the pin appears, as a consequence of allocating elements in which the pin is contained to different partitions. In the context of distributed application modelling, pin replicas are a proxy for volume of communication; the intuition is that we want to weight pins that are used by multiple elements so the algorithm is less inclined to create further replicas for it –further replicas would result in communication during runtime, the more communication the more elements that use that pin.

Thus, communication cost for each candidate partition k is calculated as follows:

$$CCost_k(pins) = \sum_{pin \in pins} \sum_{j=0}^{p} A_j(pin) \times C(k, j) \tag{6.7}$$

where adjacency $A_j(pin)$, or the number of times pin has been previously seen in partition j, accounts for pin replicas. Communication cost matrix $C(k, j)$ represents cost of communication calculated via profiling[2] between partitions k and j. Final allocation value function incorporating communication cost, instead of just overlap:

$$j \leftarrow \underset{k \in candidateParts}{\arg\max} = -CCost_k(N_e) - W(k)^{\lambda} \tag{6.8}$$

6.4.2 Quality and simulation time for architecture-aware streaming

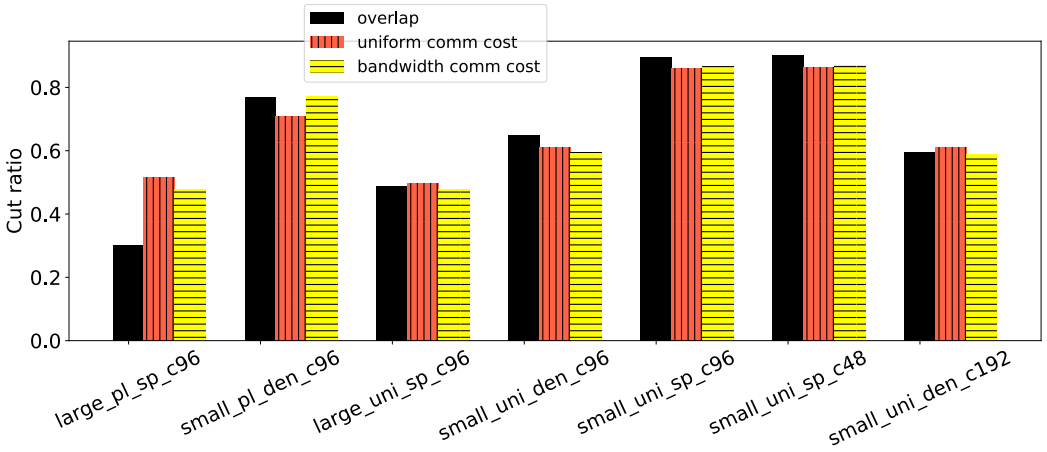
To understand the impact of using network cost of communication on parallel streaming, the new allocation value function (6.8) is compared to the previous overlap-only approach, described and evaluated in section 6.3.1. Two variants of architecture-aware streaming are used: uniform, that uses uniform costs of communication between all processes; and bandwidth communication, which uses profile-discovered peer to peer cost of communication. Note that uniform cost is not the same as overlap, since we are using now pin replica counts (0 or many), not just pin overlap (0 or 1). Figure 6.7 shows partitioning quality for all candidates. Quality metrics (hyperedge cut and SOED) provide mixed results, with no clear advantage of one strategy over another.

In order to understand the impact of architecture-awareness it is important to consider hypergraphs are models of parallel communication, hence the quality is not the primary metric. The metric we are interested in is whether the new partitioning yields better simulation times for our application —i.e. if the communication time is reduced as a consequence of the new allocation. For this, the synthetic benchmark proposed in 4.3 is used, a null-compute simulation which communication is proportional to the SOED of the hypergraph.

Figure 6.8 considers bandwidth communication costs on the same graphs as figure 6.7. Results shows that the bandwidth communication cost strategy is superior across all hypergraphs tested, with significant simulation time reductions (up to 5.2x speedup). Using uniform communication costs yields worse results than overlap

[2]Using the same approach as in 4.5.2, calculated through P2P profiling prior to streaming.

(a)



(b)

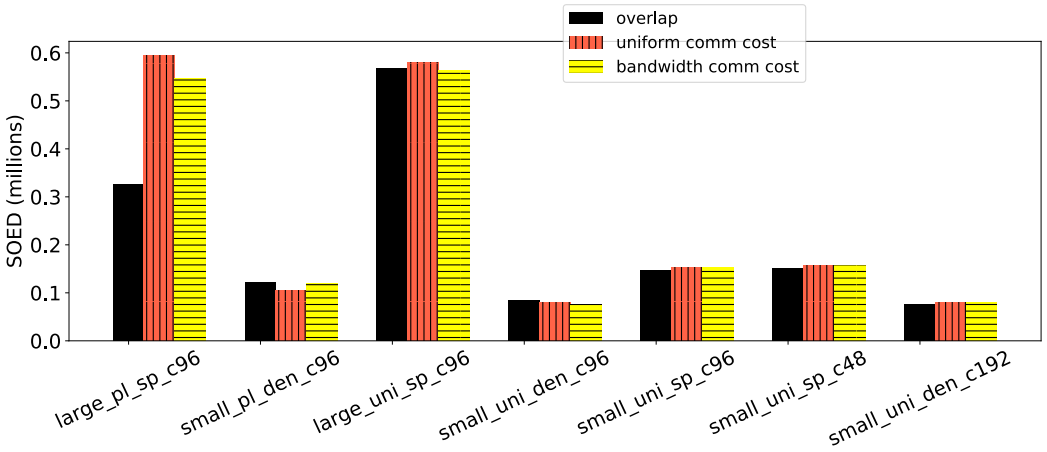
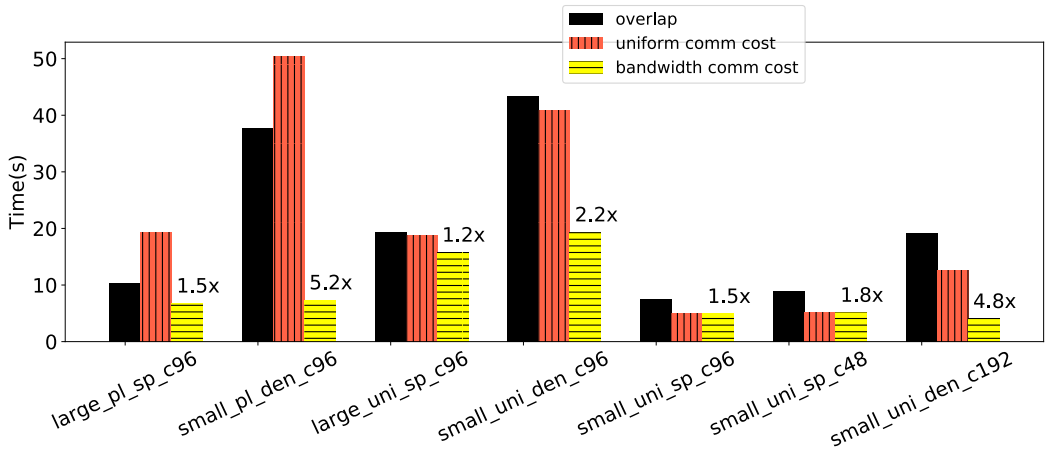


Figure 6.7: Partitioning quality (**top**: hyperedge cut, **bottom**: SOED) on a collection of hypergraphs. Target partitioning: 96 parts. Three streaming approaches that use different allocation value functions are compared: overlap only (**black**), uniform communication cost (**red**) and bandwidth communication cost (**yellow**). The quality of partitioning is comparable across alternatives, with a general degradation when using bandwidth compared to uniform communication costs. Partitioning is done using a single stream.



A

Figure 6.8: Simulation time measured on synthetic benchmark on a collection of hypergraphs. Three streaming approaches that use different allocation value functions are compared: overlap only (**black**), uniform communication cost (**red**) and bandwidth communication cost (**yellow**). Using profiled bandwidth communication costs significantly reduces the simulation time (speedup up to 5.2x). Uniform communication costs yields worse results than overlap, which demonstrates that profile bandwidth data is responsible for simulation time improvement. Partitioning is done using a single stream.

alone, which demonstrates that using profile-based communication costs is good to guide streaming partitioning algorithms in order to reduce communication cost.

To demonstrate the impact bandwidth communication cost has in allocation of workload, Figure 6.9 shows actual network communication during simulation for three selected hypergraph synthetic simulations. Total peer to peer communication for overlap and bandwidth communication cost alternatives is shown. Using bandwidth communication data results in allocations that display more communication around the central band, which is between computing nodes with fast connectivity (red band in Figure 6.9). As discussed in Figure 4.8, for an optimal utilisation of the hardware architecture, the communication activity of the parallel application should resemble the bandwidth profile. Figure 6.9 shows that this is the case for parallel streaming using bandwidth communication cost, whereas using overlap exhibits near random communication pattern. This is ultimately responsible for the improved simulation performance exhibit in Figure 6.8.

6.5 HyperPRAW partitioner

The combination of staggered parallel multistreaming, use of workload balance parameter and architecture-awareness from network communication cost forms the proposed partitioner: **HyperPRAW**. This section benchmarks HyperPRAW to state-of-the-art global hypergraph partitioning (Zoltan [62]) in the context of modelling communication in distributed applications. We continue to use the proposed synthetic simulation benchmark in section 4.3.

Figure 6.10 shows quality and simulation runtime results comparing Zoltan to HyperPRAW on a benchmark of 8 graphs. Quality results of HyperPRAW are

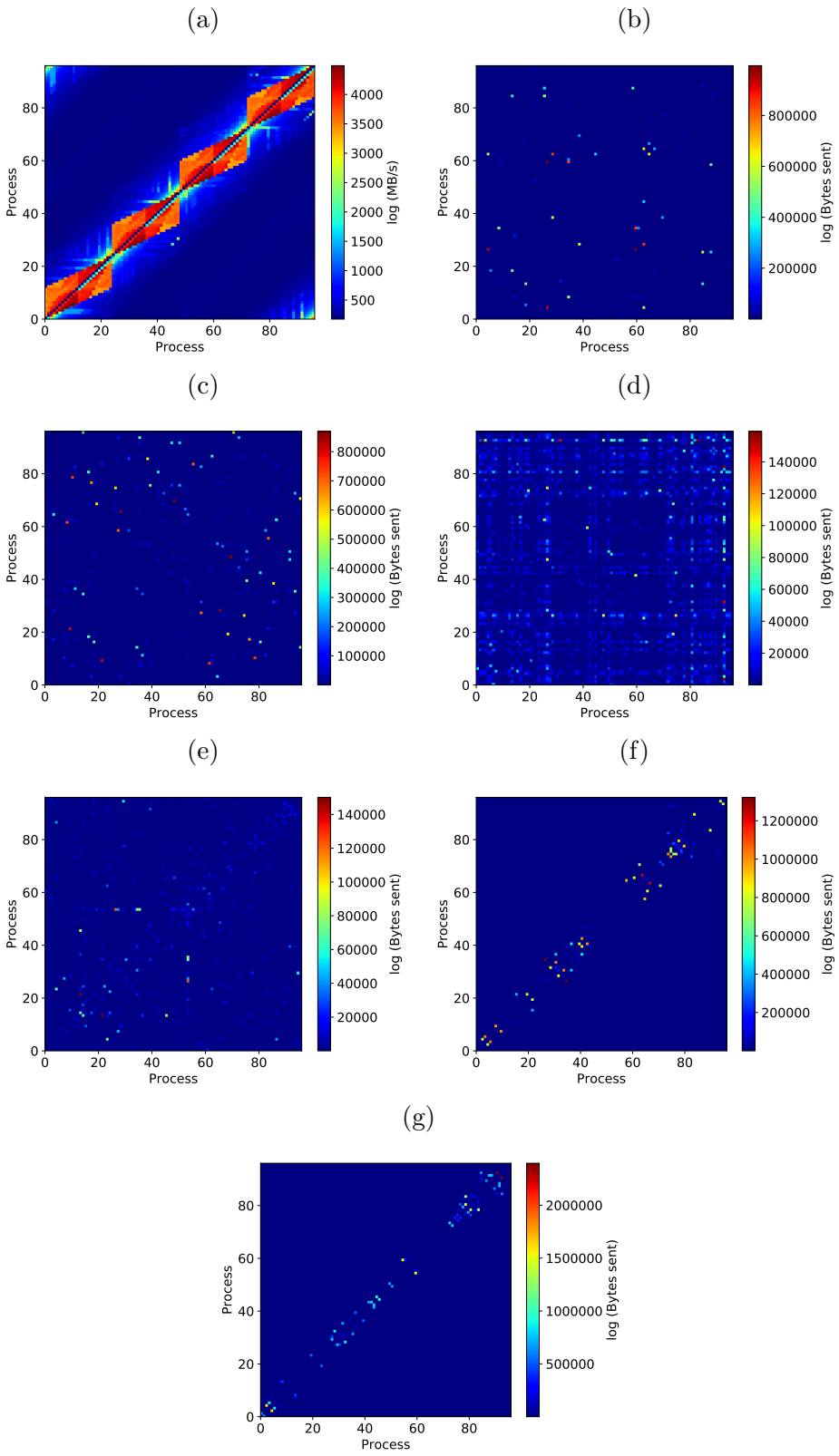
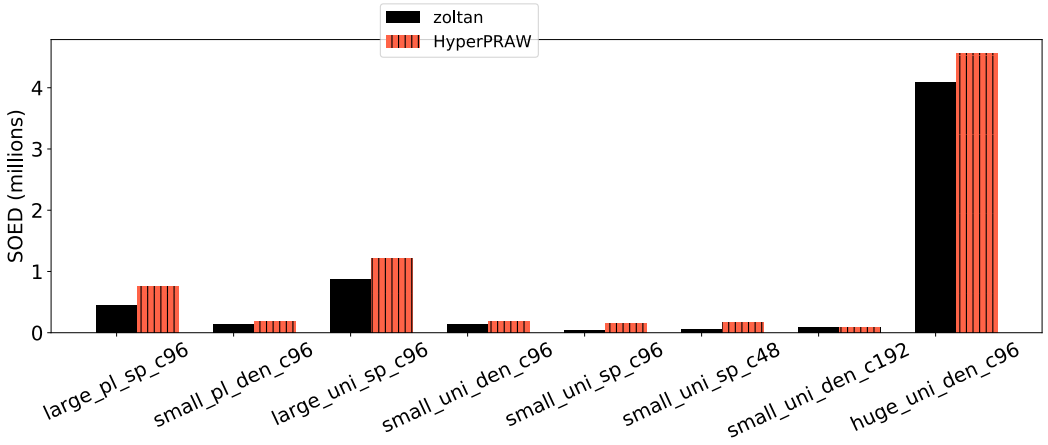


Figure 6.9: Network communication recorded during synthetic benchmark simulation. **A**: measured architecture bandwidth profile where the simulations are run (ARCHER supercomputer). Overlap streaming algorithm for graphs small uniform dense 96 (**B**), small powerlaw dense 96 (**C**) and large powerlaw sparse 96 (**D**). The same network activity is shown for streaming using bandwidth communication costs for graphs small uniform dense 96 (**E**), small powerlaw dense 96 (**F**) and large powerlaw sparse 96 (**G**). Using profiled bandwidth communication costs in partitioning results in applications that exhibit communication clustered around faster computing node pairs.

(a)



(b)

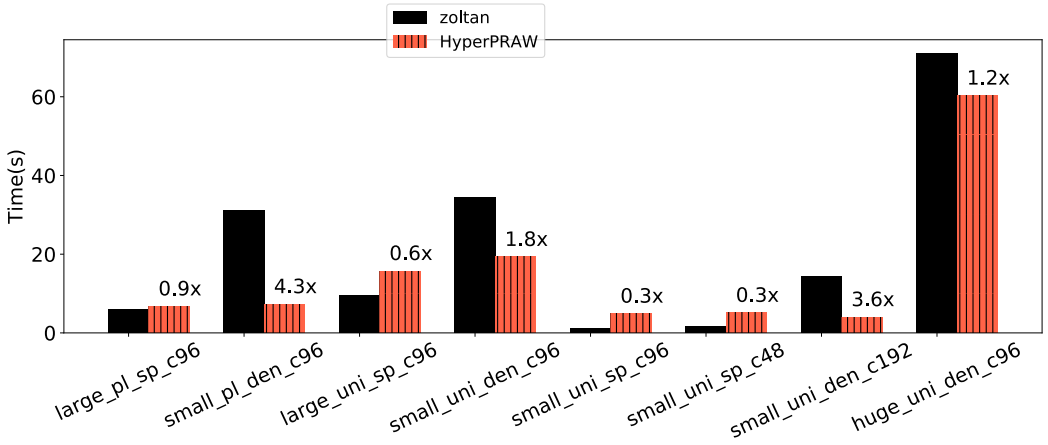


Figure 6.10: HyperPRAW vs Zoltan partitioning across benchmark hypergraphs. **A**: partitioning quality (SOED). **B**: synthetic simulation time. Even though there is significant quality loss in all partitions, HyperPRAW exhibits comparable or better simulation times in 5 out of 8 graphs (up to 4.3x), and all the dense ones. Target partitioning task size of 192 parts.

worse than Zoltan, consistent with results in the work on architecture-aware sequential streaming in chapter 4, and as demonstrated in Figure 6.7. In terms of simulation runtime, results show that HyperPRAW algorithm performs strongly on dense graphs, with speedup factors of up to 4.3x. Unsurprisingly, the streaming approach performs worse on graphs with low average cardinality (sparse), since the local information available to streams is limited and therefore it is harder to make allocation decisions that minimise global communication.

Figure 6.9 and Figure 6.10 only look at a single partitioning task of 192 partitions. To study the scalability of the approach, a series of scalability experiments are run to compare Zoltan and HyperPRAW at different parallel stream counts. Characterising scalability is important in two dimensions: strong scaling (how the algorithm reacts to partitioning the same graph using increasing number of parallel streams) and weak scaling (how the algorithm scales to bigger problems, i.e. larger partitioning tasks). The scaling experiments are done for increasing partitioning sizes (96, 192,

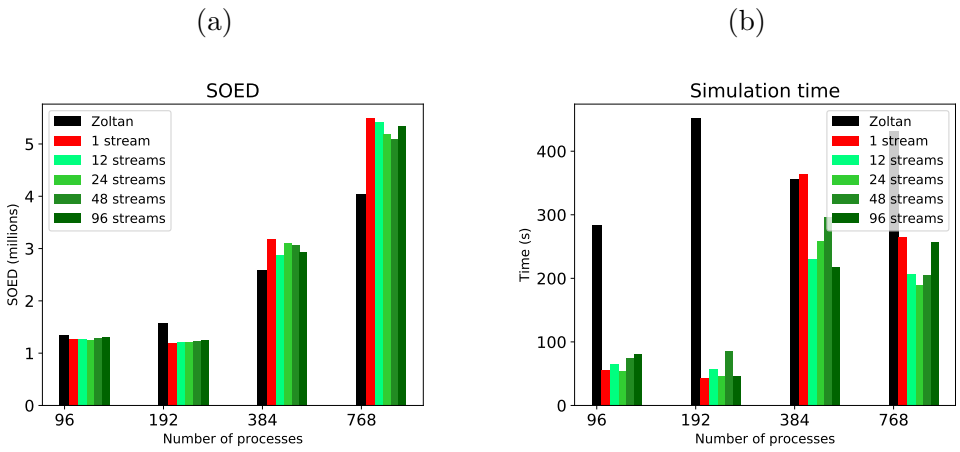


Figure 6.11: Scaling of HyperPRAW. Strong scaling is shown for each partitioning size target (96, 192, 384, 768), where the same graph is partitioned at increasing levels of parallelism (1, 12, 24, 48 ans 96 streams). Weak scaling is shown across partitioning size targets, where the same graph and at the same level of parallelism is partitioned into increasing partitioning sizes. **A**: SOED; **B**: simulation time. HyperPRAW shows good strong scaling (similar quality and simulation times using more streams). Compared to Zoltan, HyperPRAW results in significantly faster simulation times across most experiments. Hypergraph: huge uni packed 192.

384 and 768) an an extra large hypergraph —to be able to partition a graph at large part counts.

Figure 6.11 show the results of the scaling experiments. Good strong scaling for HyperPRAW, showing almost no differences between 1 and 96 streams. Similar weak scaling as Zoltan in terms of quality of partitioning (SOED). In terms of simulation runtime, HyperPRAW provides significant speedup at all partitioning sizes, with better results achieved with higher parallel stream counts.

6.5.1 Streaming memory requirements

Global partitioners such as Zoltan require the entire hypergraph to be loaded in memory during partitioning, in addition to having the graph also live in persistent storage as input. Pins associated with each vertex are expressed efficiently using Compressed Sparse Row (CSR) format. Through profiling Zoltan, the following are its memory requirements:

- *Per process*: number of local vertices (**int**), number of pins (**int**) and process id (**int**); list of partition allocation for each vertex (list of **double**, the length of the number of total vertices).
- *Global*: list of vertex ids (length of number of vertices, **double**), list of pin ids (length of number of pins, **double**), list of vertex pointers (length of number of vertices, **int**) and list of vertex weights (length of number of vertices, **float**).

Streaming partitioners do not require, in principle, full knowledge of the graph at once, since they perform allocation decisions one element at a time. Therefore, the graph does not need to reside in memory at once, nor is it needed in the harddrive at the beginning of the partitioning process. For HyperPRAW, the following data structures must be kept accessible to each stream:

- list of element (number of elements, **float**) and pin weights (number of pins, **float**)
- list of partition workload (number of partitions, **double**)
- list of partition allocation for each vertex (number of elements, **double**)
- data buffers for partition candidates (number of partitions, **double**; 3 times number of streams, **int**)
- communication cost matrix (number of partitions times number of partitions, **float**)
- seen pins (up to number of pins times number of partitions, **int**), dependent on the average replication factor for each pin –see discussion in section 6.6.

In Figure 6.12 the following assumptions are taken to calculate memory requirements: **int** and **float** 4 bytes and **double** 8 bytes (common in many architectures and programming languages). For HyperPRAW we present results using RDMA optimisation, showing only the best case implementation memory requirements where streams have access to a single central data structure for synchronisation —see section 6.6 for further discussion.

To understand the memory requirements of HyperPRAW as it scales in large graphs in real-world SNN models, Figure 6.12 shows theoretical memory requirements to partition the MVC ([195] 4.13 million neurons, 24.2 billion synapses) and the CM models ([177], 80000 neurons and 300 million synapses) with Zoltan (global approach) vs hyperPRAW (parallel streaming). There are four candidates for hyperPRAW, depending on the average replication factor chosen (average number of partitions that contain each hyperedge). The replication factor is dependent on the quality of the partitioning and hence we are providing different values as this ultimately depends on how partitionable graphs are. Values of 0.01, 0.05, 0.1 and 0.5 are chosen to be representative of reasonable partitioning results. Figure 6.12 shows that the memory requirements of HyperPRAW are significantly less than those for Zoltan, in both SNN models. The main memory contributor to HyperPRAW is the centralised data structure that holds information about what pins have been visited so far and where are they replicated. This grows as the number of streams is increased and therefore the increase in memory footprint with large partitions. Zoltan shares the entire graph amongst all processors, hence leaving most of the memory requirements independent from the number of partitions. This comparison uses a highly optimised version of HyperPRAW in which central database and data structures for partitioning are shared amongst streams —see section 6.6 for a discussion.

One important difference with HyperPRAW with respect to Zoltan is that the number of partitions (size of the partitioning task) and the number of parallel streams used are independent. Because the memory footprint of HyperPRAW grows primarily with the number of streams (not the size of the partitioning task), one can always limit the number of streams used to fit the hardware available and still be able to partition large graphs. This would not be possible with global partitioners since the entire graph must reside in memory, independent of the level of parallelism.

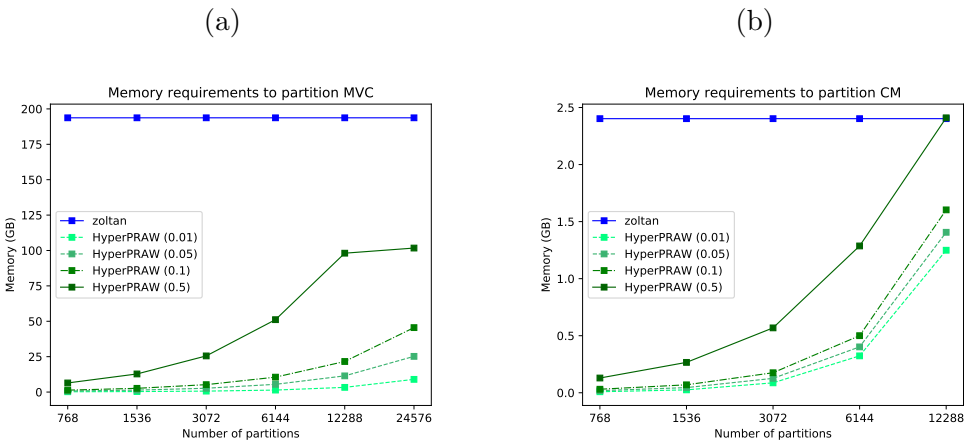


Figure 6.12: Memory requirements for Zoltan vs HyperPRAW when partitioning the MVC model (**A**) and CM model (**B**). There are four candidates for hyperPRAW, depending on the average replication factor chosen. For HyperPRAW, the number of processes indicate the size of the problem (number of partitions) and the parallel scale (number of streams). Due to the streaming nature of HyperPRAW, graphs do not have to reside in memory and therefore memory requirements are significantly lower than for Zoltan. The average pin replication is calculated as the chosen replication factor times the number of partitions. The value is then capped to a maximum of the number of pins over number of elements (number of pin replicas cannot exceed the average cardinality, or number of elements containing a pin), which explains the plateau seen in HyperPRAW (0.5) in MVC.

6.5.2 Partitioning performance

HyperPRAW has been shown to be effective in modelling distributed applications to reduce simulation time, and doing so with far less memory requirements than global partitioners. Figure 6.13 shows scaling of partitioning time in hyperPRAW with more number of streams vs Zoltan. HyperPRAW maximum parallelism shown is up to 96 streams, whereas Zoltan runs in parallel across all computing nodes. Therefore the only one that is an accurate comparison is when the partitioning size is 96, in which the partitioning times are similar. In terms of weak scaling, HyperPRAW scales well as the partitioning problem increases (higher number of partitions). Partitioning with more streams (strong scaling) does not seem to have an impact on the time it takes to partition the hypergraph. This is likely due to the increased communication demands for synchronising parallel streams. See discussion in section 6.6 on the possibility of using a centralised data structure, instead of a synchronisation-based parallelisation.

6.6 Discussion

Global hypergraph partitioners require the hypergraph to be fully loaded in memory. State of the art multilevel partitioners struggle to scale to large graphs, with the coarsening and refinement phases acting as a bottleneck [49, 125]. Therefore, streaming algorithms are an alternative to tackle those limitations as they only required local, streamed (one element at a time) information.

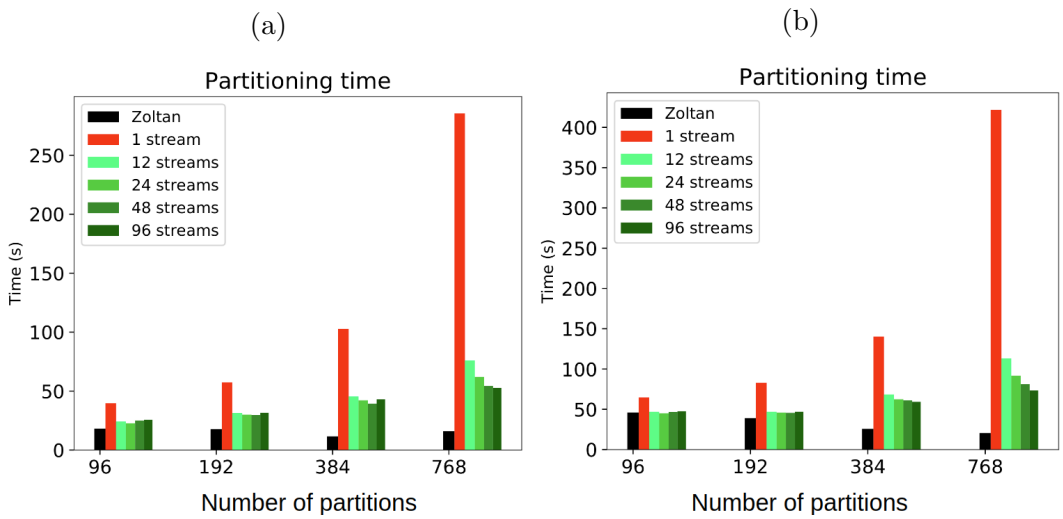


Figure 6.13: Partitioning time for HyperPRAW vs Zoltan synthetic hypergraphs. **A**: huge uni den 96, **B** huge uni packed 192. Both plots show partitioning times at different partition sizes (number of partitions) at increasing parallel levels (number of streams). HyperPRAW displays on par performance with Zoltan at its maximum parallelism level (96 partitions / streams). Good weak scaling shown by HyperPRAW, with comparable times of partitioning with an increased task (higher number of partitions). Poor strong scaling, likely due to the increased communication demands of parallel synchronisation with high number of streams.

Previous approach to hypergraph streaming partitioning [6] is limited in its applicability to large graphs as it is sequential. Our approach in chapter 4 is sequential, suffering from the same performance limitations.

The work in this chapter demonstrates that a staggered allocation start helps mitigate partitioning quality degradation when scaling to high stream counts (greater than 16), resulting in partitions with higher Cluster Recoverability. A staggered approach makes each stream start evaluating candidate partitions at a different point, which helps to avoid overcrowding initial partitions at the start of the process. This has been shown to degrade the quality of partition since future allocations will be more likely to continue to place elements there, potentially under utilising other partitions. Employing a parameterised workload imbalance factor within the allocation value function further increases inter-process work balance (Figure 6.6), increase Cluster Recoverability (between 5-15%) and reduce loss of partitioning quality (Figure 6.5) in multistreams. A new parameter, λ, is introduced to control the weight of workload imbalance in the allocation value function. For all the graphs in this chapter an optimal value seems to lay between 0.5 and 1.0. All results shown use a handpicked λ value that minimises workload imbalance (found during exploratory search by restreaming with different values to find the value that leads to sufficient imbalance between maximum and average workload). Finding adequate λ values based on hypergraph features is left as future work.

An alternative approach to fixed staggered start to avoid overcrowding partitions is to start evaluating partition candidates at a random point for each stream and element. In practice, using a workload imbalance factor in the allocation function is likely to mimic some of the benefits of a random restart (favouring less filled partitions), and a careful evaluation is left as future work.

When using hypergraphs to model communication in distributed applications, chapter 4 demonstrated that including network bandwidth data in the allocation value function leads to reduced application runtime by redirecting heavy and frequent communication via faster peer to peer links. This chapter shows that HyperPRAW, a parallel multistream approach, also benefits from using profile-based network communication costs, with runtime speedups of up to 5.2x (Figure 6.8) on synthetic benchmark simulations. The resulting simulation communication pattern resembles the network peer to peer bandwidth graph (Figure 6.9), which is responsible for the increased performance.

Compared to Zoltan (state-of-the-art global partitioner), HyperPRAW shows significant runtime speedup on hypergraphs with high average cardinality (up to 4.3x in a synthetic simulation benchmark, Figure 6.10). High cardinality allows the algorithm to leverage the history of previously seen pins ($seenPins_{pin,j}$ in algorithm 3) and make better allocation decisions to reduce the replication of pins. HyperPRAW shows good strong scaling, with minimal quality loss or runtime simulation performance degradation with 1, 12, 24, 48 and 96 streams (Figure 6.11). This results demonstrate that HyperPRAW is an effective parallel multistream hypergraph partitioning approach with minimal loss as the algorithm scales to higher stream counts.

For power-law graphs [166] demonstrates that distributing the coarsening phase of a multilevel partitioning algorithm is close to require to have global knowledge of the graph for each parallel worker, which makes it impractical for very large graphs. Therefore it is important that the memory requirements of a partitioner are within reason. Figure 6.12 shows that HyperPRAW memory footprint is far less than loading the entire graph in memory. Two real world hypergraphs representing Spiking Neural Networks (CM and MVC models) show memory requirements that scale with the average pin replication factor (i.e. how well the partitioner clusters elements without splitting them) and the number of parallel streams. In all cases, memory requirements are significantly less than loading the entire graph. This can also serve as a tool to detect overpartitioning for a model. If memory requirements scale poorly in Figure 6.12, it may be an indication that we are over parallelising the partitioning. Note that global parallel partitioners such as Zoltan cannot dissociate the size of the partitioning problem (number of partitions) and the level of parallelism. HyperPRAW treats these as different input parameters and therefore offers flexibility to limit the level of parallelism if it surpasses the capacity of the running system (i.e. still split the hypergraph in high number of partitions whilst reducing the number of streams employed to do so).

A naive implementation of a multistreaming algorithm has all required data structures replicated per stream. However, there are optimisation tweaks that can be made to reduce the memory footprint. In cluster computers, one can share data structures amongst processes that have access to shared memory (in ARCHER, every 24 processes). In addition, modern architectures allow access to non-shared memory spaces, such as Remote Direct Memory Access (RDMA [3]) which further reduces the number of replications to one for the entire algorithm. We have only implemented the first level of optimisation (shared-memory access) but on architectures that are RDMA compatible the second level should be trivial to implement. Replicating data structures only on non-shared memory spaces requires, in ARCHER, one copy

[3] `http://www.rdmaconsortium.org/`

of the central data structure every 24 processes (since 24 processes share memory space within the same computing node); hence for 768 parallel streams, there is a redundancy of 32 copies.

In terms of partitioning computational performance, Figure 6.13 shows HyperPRAW has good weak scaling: simulation time scales linearly with the size of the problem. HyperPRAW shows significant speedup when scaling from 1 to 12 streams but further parallelism has minimal impact, likely due to increased synchronisation during the partitioning process between more streams. Further work to improve this may include: using window-based synchronisation to reduce the number of synchronisation events, use of central data structures and RDMA architectures which bypass the need for explicit synchronisation.

Across all experiments, HyperPRAW has assumed uniform element and pin weights. However, HyperPRAW offers support for element and pin weighing, by using multiplier factors associated with element and pin ids. In the context of modelling distributed applications, element weights can be used to model irregular computations, which would reflect more accurate workload balance if elements in fact do carry out different computation sizes. Likewise, pin weights can be used to control the importance of a hyperedge, or the volume of communication associated with each hyperedge. This is in line with previous work on modelling message volume and count [197, 213] and modelling skewed graph traversals [78].

Chapter 6 demonstrated that graph partitioning and sparse exchange communication patterns significantly reduce overhead of communications in SNN simulations. HyperPRAW can further the improvement by allowing the neuron allocation process to be architecture-aware, placing frequently communicating neurons in clusters with high bandwidth. This chapter has shown the memory requirements of HyperPRAW for two SNN models: CM and MVC, displaying better scalability than global partitioners. Thus, HyperPRAW is a suitable candidate to replace workload allocation algorithms in SNN simulations to reduce communication overhead and enable better scaling. Current SNN simulators do not offer the flexibility to test SNN simulations distributed using HyperPRAW, since they do not allow custom neuron allocation process or custom distributed communication patterns. This justified building a new SNN test-bed framework to test our hypothesis in section 3.3. Unfortunately the test-bed simulator does not apply memory optimisations that have been proposed in state-of-the-art SNN simulators over the years [136, 137, 124], as the purpose is to validate the impact of workload allocation and communication strategies, not to undertake the huge task of designing a feature complete simulator. To fully benefit from the optimisations brought by HyperPRAW in large scale distributed models, SNN simulators must allow for custom workload allocation procedures to be used.

6.7 Related work

Hypergraph partitioning and modelling application communication as hypergraphs has been identified as a good approach that can help mimimise communication cost in distributed application problems [197, 213, 44, 18, 19, 5, 194]. But they are limited to using off-the-shelf hypergraph partitioning and focus their efforts on modelling their constraints as part of the hypergraph. HyperPRAW demonstrates that those constraints can be effectively made part of the partitioning process itself.

Our focus is not on graph streaming, but on hypergraph streaming. For a survey

on graph streaming, see [156, 1]. Converting a hypergraph into a graph has been proposed, but cannot be done without compromises [117, 122].

GraSP [21] proposes a parallel streaming approach but it is only applicable to graphs and only considers overlap and workload. Our approach is based on GraSP, but with key differences: 1) it is applicable to hypergraphs; 2) there is frequent synchronisation of workload and partition assignments between streams to preserve high quality partition metrics; 3) it mitigates the impact on workload that a multistream has using staggered allocation; 4) it includes network bandwidth in the allocation value function. HoVerCut [189] is a parallel graph streaming partitioning that uses a central state that is updated every few iterations between streams.

Alistarh et al. [6] proposes the only streaming hypergraph partitioner known to date. It is sequential and limited to only consider pin overlap as the only factor for element allocation (architecture-agnostic). The results in this chapter demonstrate that this can be improved upon by using network bandwidth costs during element allocation and with the use of a workload parameter.

Mapping processes to resources to reduce communication has been done before [112, 121], but assumes process workload allocation has been done already. This is a coarser level of granularity than considering individual computing and communicating elements and therefore limits the amount of communication optimisation possible.

Two domains that rely on efficient processing and representation of large graphs are graph processing applications (such as social media analysis) and graph embedding frameworks (learning efficient representations of large graphs). Prominent frameworks have shown the importance of graph partitioning to tackle bigger-than-memory graphs [141, 222, 145] but have been limited to using off-the-shelf architecture-agnostic partitioners. Using architecture-aware HyperPRAW could further improve the performance of those algorithms at large distributed scale.

6.8 Conclusion

The work in this chapter has presented HyperPRAW, an efficient architecture-aware parallel streaming hyperpgraph partitioning algorithm implementation that overcomes the limitations of sequential streaming partitioners (low performance) and global partitioners (high memory requirements). The results of this chapter are:

- Characterised parallel streaming performance contributors: staggered streaming start to improve partitioning quality enhances quality metrics (AR, CR, SOED); and the use of workload balance parameter increases quality of partitioning by 5-15% better AR, with less quality loss when scaling to multiple streams (contribution C11).

- Improved synthetic simulation runtime performance by up to 5.2x speedup using network bandwidth communication-based allocation function over overlap-only streaming [6] (contribution C12).

- Faster synthetic simulation runtime (up to 4.3x speedup) compared to global state-of-the-art global hypergraph partitioners on graphs with high average cardinality (contribution C12).

- Reduced memory requirements against global partitioners (over an order of magnitude on real world SNN models), making HyperPRAW capable of handling larger hypergraphs (contribution C13).

6.9 Further work

This chapter shows an effective implementation of an architecture-aware parallel multistreaming hypergraph partitioning algorithm. The focus has been on characterising the features of the algorithm, such as parallel scalability, use of workload and network bandwidth parameters, staggered vs uniform streaming start; and how those features impact distributed applications modelled as hypergraphs. Future areas of research may include:

- Explore λ workload parameter and see dependencies with graph attributes
- Explore alternatives to overlap allocation value function, in particular those that have been shown to work well with power-law graphs (High Degree Replication First, HDRF [173, 189]).
- Explore parallel optimisations such as use of central data structures to reduce synchronisation events with RDMA-enabled architectures, or window-based streaming [169] (batching elements before synchronising).
- Experiment with variable workload imbalance tolerance. Instead of fixing it to an arbitrary value, show how this value impacts overall performance (with non null-compute simulations).

Chapter 7

Conclusion

In the last decade there has been a growing scientific focus on computational neuroscience as a means to learn about the brain and its functions, with multiple international projects setting out to further our understanding of the brain [150, 7, 133, 175]. In that journey, the simulation of large-scale models of neural networks are key to scientific discovery. Due to the size and complexity of the brain, the computing resources required at the brain scale simulation far surpass the capabilities of single computers today. Unfortunately, indefinitely increasing the speed of single computing processors is no longer an option. Two factors contribute to that: physical limitations on heat dissipation (*power wall effect* [12]), and the slow-down of energy efficiency scaling [99, 151]). Instead, the hardware trend to improve processor speed is to increase the number of computational cores, with large distributed systems commonly used in HPC systems. Scaling applications to efficiently run on distributed computation model is non-trivial and brings new challenges, mainly the extra communication overhead imposed by dividing workload into a mix of shared and non shared memory computing nodes. This communication becomes a bottleneck and limits scalability.

This research aimed to *propose effective architecture-aware strategies with hypergraph models and partitioning algorithms to increase scalability of large distributed applications in High Performance Computing (HPC) systems*, of which SNN simulations are a typical example. Based on quantitative profiling and experimentation, it can be concluded that targeting the communication phase of distributed applications leads to significant runtime speedup when considering the network bandwidth during workload allocation. To that end, this work has proposed and characterised a novel architecture-aware parallel streaming partitioning algorithm that successfully models distributed application execution to minimise communication overhead resulting in faster application execution time.

Chapter 3 characterises the communication overhead as a bottleneck in scaling Spiking Neural Network (SNN) simulations. The three phases of communication (implicit synchronisation, process handshake and data exchange) are shown to contribute to most of the execution time during simulation when running in > 6000 distributed computing units (over 60% of total simulation time). The chapter shows the potential of using hypergraphs to model SNN simulations, and hypergraph partitioning to optimise neuron allocation to computing nodes to minimise communication. By itself, graph partitioning increases the communication sparsity between computing nodes (less than 90% Average Runtime Neighbour processes), but it is

not enough to improve runtime performance with commonly use communication patterns such as personalised census (PEX), which are highly inefficient for sparse communications. Making use of the increased sparsity, sparsity-aware Dynamic Sparse Data Exchange communication collective (NBX) is able to significantly reduce implicit synchronisation time (up to 90%), a measurement of communication imbalance. The synergy between hypergraph partitioning (which increases sparsity) and NBX communication (which efficiently handles sparse communication) significantly reduce volume data exchanged (up to 80%) and simulation time (up to 73%) on a large scale SNN model running over HPC systems. SNN simulators to date lack customisation on the neuron allocation processes (round-robin, random) or communication algorithm (all to all, PEX), which makes it hard to benchmark alternative allocation or communication strategies. To facilitate benchmarking in SNN simulations, the chapter proposes a novel framework for testing communication and workload allocation strategies Spiking Neural Network simulations.

HPC systems show highly hierarchical, non-uniform communication links, with communication speed and latency varying wildly in between computing nodes. Routing distributed applications communication patterns through faster communication links would increase overall simulation performance. Having proved that hypergraph partitioning allocation of computing elements in SNN improves computational efficiency in simulations, Chapter 4 presents a novel hypergraph partitioning algorithm that incorporates network bandwidth cost in its allocation function. The results demonstrate that making use of bandwidth communication cost leads to application communication patterns that resemble the hierarchical network bandwidth costs, routing frequent communication elements through fast computing node links. This leads to significantly reduced simulation times (up to 14x speedup compared to architecture-agnostic partitioners). The chapter proposes an application-agnostic synthetic benchmark to evaluate communication costs in runtime for applications modelled as hypergraphs, which goes beyond SNN-specific simulation and can help assess workload and communication in other distributed applications.

Working with hypergraph partitioners and hypergraphs to model distributed applications, it is important to be able to benchmark algorithms on a collection of hypergraphs that exhibit the range of features that the application themselves possess. For instance, many real-world graphs exhibit power-law degree distributions ([142, 70, 39]). An algorithm that aims at allocating work on such a graph would need to be evaluated on graphs that have that distribution. To date, there are no hypergraph generation algorithms that are sophisticated enough to specify features such as vertex degree and hyperedge cardinality distributions, number of clusters, average cardinality and graph size. Chapter 5 proposes a novel parametric hypergraph generator that can be used to automatically generate hypergraphs with custom features, facilitating benchmarking in tasks that involve hypergraph models such as workload allocation in distributed applications. In particular it can generate a class of hypergraphs that represent common features in neural simulation.

The architecture-aware sequential streaming hypergraph partitioner proposed in chapter 4 is effective in improving distributed simulation performance. However, there are strong performance limitations that reduce its applicability to large hypergraphs: being sequential and with several passes, it is slow and requires full knowledge of the graph. State-of-the-art global multilevel partitioners produce good quality partitions, but have high memory requirements (the entire graph must fit in

memory) and have been shown to struggle to scale [49, 125]. Chapter 6 proposes and characterises a novel architecture-aware, parallel multistreaming hypergraph partitioner (HyperPRAW) that tackles all the limitations above, and demonstrates that it produces consistent speedup over naive streaming approaches that only consider hyperedge overlap (up to 5.2x speedup). Compared to global partitioners in dense hypergraphs, HyperPRAW is able to produce workload allocations that result in speedup runtime in synthetic simulation benchmark (up to 4.3x), even with a significant degradation in quality metrics (which are not representative of the outcome of partitioning when it is used to model distributed applications in runtime systems). HyperPRAW has the potential to scale to very large hypergraphs as it only requires local information to make allocation decisions, with an order of magnitude less memory footprint than global partitioners.

The work in this thesis has the following scope limitations which sets up avenues for future work:

- SNN simulations have been used as a case in point for complex system simulations that requires scaling to large scale distributed systems.
- Dynamic application runtime communication and computation patterns have not been considered during the allocation process. Gathering real time application-specific data during simulation could be considered to further improve the mapping of hardware capabilities and distributed application patterns.
- Focus on one distributed computational model (CPU in shared and non-shared memory settings), which is the most common scenario in HPC systems. The algorithms could be extended to other computational models such as GPU farms or vector processors.
- All communication, profiling and benchmarking in the modelled distributed application between computing nodes is done using MPI, the *lingua franca* of HPC systems. Other systems also used in HPC could be considered for optimisation over heterogeneous architectures: NVLink for NVIDIA GPU to GPU, OpenMP and OpenACC for multithreaded shared memory communication.
- In a cloud computing distributed environment, communication cost goes beyond runtime to include a monetary dimension. Models that minimise not only communication volume, but also cost, could have a big impact in cloud computing users. Extending the proposed partitioners to include economic parameters can help in this space.

HyperPRAW is a suitable candidate to replace workload allocation algorithms in SNN simulations to reduce communication overhead and enable scaling to larger models. Current SNN simulators do not offer the flexibility to test simulations with sophisticated neuron allocation algorithms such as HyperPRAW. Thus, to fully benefit from the optimisations brought by HyperPRAW in large scale distributed models, it is recommended that SNN simulators allow for custom workload allocation procedures to be employed. On the back of the reduced inter-process connectivity brought by partitioning algorithms, communication can be made more efficient with the use of sparse data exchange collectives. A second recommendation to the SNN simulation community is to enable research on the field by allowing custom communication patterns to be used.

In summary, this thesis proposes a novel, parallel, scalable, streaming hypergraph partitioning algorithm (HyperPRAW) that can be used to help scale large distributed simulations in HPC systems. HyperPRAW addresses three of the main scalability challenges: it produces *highly balanced distributed computation and communication*, minimising idle time between computing nodes; it *reduces the communication overhead* by placing frequently communicating simulation elements close to each other, where the communication cost is minimal; and it provides a solution with a *reasonable memory footprint* that allows tackling larger problems than state-of-the-art alternatives such as global multilevel partitioning.

HyperPRAW achieves better distribution of workload by reducing the communication costs that limit scalability of distributed applications such as SNN simulations to large scale distributed HPC systems. This key contribution enables bigger and more complex models to be simulated, pushing the field one step closer to whole-brain simulations.

Bibliography

[1] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov. Streaming Graph Partitioning: An Experimental Study. In *Proceedings ofthe VLDB Endowment*, volume 11, pages 1590–1603, Rio de Janeiro, Brazil, 2018.

[2] M. Abduljabbar, G. Markomanolis, H. Ibeid, R. Yokota, and D. Keyes. Communication Reducing Algorithms for Distributed Hierarchical N-Body Problems with Boundary Distributions. In *ISC 2017: High Performance Computing*, pages 79–96. Springer Link, 2017.

[3] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS'06 Proceedings of the 20th international conference on Parallel and distributed processing*, pages 124–124, Rhodes Island, Greece, 2006.

[4] T. Adachi, N. Shida, K. Miura, S. Sumimoto, A. Uno, M. Kurokawa, F. Shoji, and M. Yokokawa. The design of ultra scalable MPI collective communication on the K computer. *Computer Science - Research and Development*, 28(2-3):147–155, 2012.

[5] K. Akbudak and C. Aykanat. Simultaneous input and output matrix partitioning for Outer-product-parallel sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 36(5):568–590, 2014.

[6] D. Alistarh and J. Iglesias. Streaming Min-max Hypergraph Partitioning. *Advances in Neural Information Processing Systems 28 (NIPS 2015)*, pages 1–17, 2015.

[7] A. P. Alivisatos, M. Chun, G. M. Church, R. J. Greenspan, M. L. Roukes, and R. Yuste. The Brain Activity Map and Functional Connectomics. *Neuron*, 6(January 2012):1–42, 2012.

[8] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D S Modha. The cat is out of the bag: cortical simulations with 10ˆ9 neurons, 10ˆ13 synapses. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, number c, pages 1–12, Portland, Oregon (USA), 2009.

[9] R. Ananthanarayanan and D. S. Modha. Anatomy of a cortical simulator. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC '07*, number c, page 1, Reno, Nefvada (USA), 2007.

[10] R. Ananthanarayanan and D. S. Modha. Scaling, stability and synchronization in mouse-sized (and larger) cortical simulations. *BMC Neuroscience*, 8(Suppl 2):P187, 2007.

[11] A. Antelmi, G. Cordasco, and C. Spagnuolo. On Evaluating Graph Partitioning Algorithms for Distributed Agent Based Models on Networks. In *Euro-Par 2015: Parallel Processing Workshops*, pages 367–378, 2015.

[12] K. Asanovic, B. C. Catanzaro, D. Patterson, and K. Yelick. The Landscape of Parallel Computing Research : A View from Berkeley. Technical report, University of California, 2006.

[13] C. Augonnet and R. Namyst. A unified runtime system for heterogeneous multicore architectures. In *2nd Workshop on Highly Parallel Processing on a Chip (HPPC 2008)*, Las Palmas de Gran Canaria, 2008.

[14] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008.

[15] M. Badoual, Q. Zou, A. P. Davison, M. Rudolph, T. Bal, Y. Frégnac, and A. Destexhe. Biophysical and phenomenological models of multiple spike interactions in spike-timing dependent plasticity. *International journal of neural systems*, 16(2):79–97, 2006.

[16] R. Bajaj and D. P. Agrawal. Improving Scheduling of Tasks in a Heterogeneous Environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, 2004.

[17] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI on a million processors. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 LNCS, pages 20–30, Berlin, Heidelberg, 2009.

[18] G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo. Communication optimal parallel multiplication of sparse random matrices. *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures - SPAA '13*, (2):222, 2013.

[19] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz. Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication. *arXiv:1603.05627v1 [cs.DC]*, 2016.

[20] G. Barlas. *Multicore and GPU Programming.* 2015.

[21] C. Battaglino, P. Pienta, and R. Vuduc. GraSP: Distributed Streaming Graph Partitioning. *Proceedings of the 1st High Performance Graph Mining Workshop - HPGM '15*, 2015.

[22] M. Beyeler, K. D. Carlson, T. S. Chou, N. Dutt, and J. L. Krichmar. CARLsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, volume September, Killarney, 2015.

[23] A. Bhatelé, G. R. Gupta, L. V. Kale, and I. H. Chung. Automated mapping of regular communication graphs on mesh interconnects. *17th International Conference on High Performance Computing, HiPC 2010*, 2010.

[24] A. Bhatelé, L. V. Kale, and S. Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *Proceedings of the 23rd international conference on Conference on Supercomputing - ICS '09*, page 110, 2009.

[25] B. Bilel, N. Navid, and M. S. Bouksiaa. Hybrid CPU-GPU distributed framework for large scale mobile networks simulation. *Proceedings - IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 44–53, 2012.

[26] A. Binotto, C. E. Pereira, A. Kuijper, A. Stork, and D. W. Fellner. An Effective Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi and Many-Core Desktop Platforms. *2011 IEEE International Conference on High Performance Computing and Communications*, (September):78–85, 2011.

[27] K. R. Bisset, J. Chen, X. Feng, V. S. Kumar, and M. V. Marathe. EpiFast: a fast algorithm for large scale realistic epidemic simulations on distributed memory systems. *Proceedings of the 23rd International Conference on Supercomputing*, pages 430–439, 2009.

[28] I. Blundell, R. Brette, T. A. Cleland, T. G. Close, D. Coca, A. P. Davison, S. Diaz-Pier, C. Fernandez-Musoles, P. Gleeson, D F. M. Goodman, M. Hines, M. W. Hopkins, P. Kumbhar, D. R. Lester, B. Marin, A. Morrison, E. Müller, T. Nowotny, A. Peyser, D. Plotnikov, P. Richmond, A. Rowley, B. Rumpe, M. Stimberg, A. B. Stokes, A. Tomkins, G. Trensch, M. Woodman, and J. M. Eppler. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Frontiers in Neuroinformatics*, 12, 2018.

[29] K. A. Boahen. Point-to-point connectivity between neuromorphic chips using address events. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(5):416–434, 2000.

[30] S. H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, C-30(3):207–214, 1981.

[31] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[32] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science and Engineering*, 15(6):36–45, 2013.

[33] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. J. Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[34] B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for MPI communication optimization. *Computers and Fluids*, 80(1):372–380, 2013.

[35] R. Brette and D. Goodman. Vectorized algorithms for spiking neural network simulation. *Neural computation*, 23(6):1503–1535, 2011.

[36] R. Brette and D. Goodman. Simulating spiking neural networks on GPU. *Network Computation in Neural systems*, 23(4):167–82, 2012.

[37] R. Brette, M. Rudolph, T. Carnevale, M. L. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. Harris, M. Zirpe, T. Natschläger, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. El Boustani, and A. Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, 23(3):349–398, 2007.

[38] W. M. Brown, M. Yamada, S. J. Plimpton, and A. N. Tharrington. Implementing molecular dynamics on hybrid high performance computers - Three-body potentials. *Computer Physics Communications*, 184(12):2785–2793, 2013.

[39] E. Bullmore and O. Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(4):312–312, 2009.

[40] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. *Lecture Notes in Computer Science*, 9220:117–158, 2013.

[41] D. V. Buonomano. Decoding temporal information: A model based on short-term synaptic plasticity. *The Journal of neuroscience*, 20(3):1129–41, 2000.

[42] N. T. Carnevale and M. L. Hines. *The Neuron Book*. Cambridge University Press, Cambridge (UK), 2006.

[43] A. Cassidy, A. G. Andreou, and J. Georgiou. Design of a one million neuron single FPGA neuromorphic system for real-time multimodal scene analysis. *45th Annual Conference on Information Sciences and Systems, CISS 2011*, pages 1–6, 2011.

[44] U. V. Catalyurek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 10(7):673–693, 1999.

[45] U. V. Catalyurek and C. Aykanat. PaToH: Partitioning Tool for Hypergraphs. Technical report, 1999.

[46] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. Heaphy, and L. A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. *Proceedings - 21st International Parallel and Distributed Processing Symposium, IPDPS 2007; Abstracts and CD-ROM*, pages 1–11, 2007.

[47] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen. A repartitioning hypergraph model for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 69(8):711–724, 2009.

[48] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12*, pages 1–13, 2012.

[49] T. Chen and B. Li. A distributed graph partitioning algorithm for processing large graphs. In *Proceedings - 2016 IEEE Symposium on Service-Oriented System Engineering, SOSE 2016*, pages 71–77, Oxford, UK, 2016.

[50] K. Cheung, S. R. Schultz, and W. Luk. NeuroFlow: A general purpose spiking neural network simulation platform using customizable processors. *Frontiers in Neuroscience*, 9(JAN):1–15, 2016.

[51] K. Chow, Y. Kwok, H. Jin, and K. Hwang. Comet: A Communication-Efficient Load Balancing Strategy for Multi-Agent Cluster Computing. In *Proceedings of ParCo'99*, 1999.

[52] K. Chronaki, A. Rico, R.M. Badia, and E. Ayguadé. Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015.

[53] F. Chung and L. Lu. The average distance in a random graph with given expected degrees. *Proceedings of the National Academy of Sciences*, 99(25):15879–15882, 2004.

[54] A. Condon and R. M. Karp. Algorithms for Graph Partitioning on the Planted Partition Model. *Random Structures and Algorithms*, 18:116–140, 2001.

[55] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo. Bringing together efficiency and effectiveness in distributed simulations: The experience with D-Mason. *Simulation: Transactions of the Society for Modeling and Simulation International*, 89(10):1236–1253, 2013.

[56] G. Cordasco, A. Mancuso, F. Milone, and C. Spagnuolo. Communication Strategies in Distributed Agent-Based Simulations : The Experience with Communication Strategies in Distributed Agent-Based Simulations : The Experience with D-M ASON. *Euro-Par 2013 Workshops*, (January):533–543, 2014.

[57] H. Cornelis and E. De Schutter. Neurospaces:Towards automated model partitioning for parallel computers. *Neurocomputing*, 70(10-12):2117–2121, 2007.

[58] M. Culpo. Current bottlenecks in the scalability of OpenFOAM on massively parallel clusters. Technical report, 2011.

[59] G. D'Angelo. Parallel and distributed simulation from many cores to the public cloud. *Proceedings of the 2011 International Conference on High Performance Computing and Simulation, HPCS 2011*, (July 2011):14–23, 2011.

[60] A. P. Davison, M. L. Hines, and E. Muller. Trends in programming languages for neuroscience simulations. *Frontiers in Neuroscience*, 3(3):374–380, 2009.

[61] M. Deveci, K. Kaya, B. Ucar, and U. V. Catalyurek. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *Journal of Parallel and Distributed Computing*, 77:69–83, 2015.

[62] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006:1–15, 2006.

[63] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2-3 SPEC. ISS.):133–152, 2005.

[64] M. Diener, E. H. M. Cruz, M. A. Z. Alves, M. S. Alhakeem, P. O. A. Navaux, and H. U. Heiß. Locality and balance for communication-aware thread mapping in multicore systems. In *Euro-Par 2015: Euro-Par 2015: Parallel Processing*, volume 9233, pages 196–208, 2015.

[65] M. Diesmann and M. Gewaltig. NEST: An environment for neural systems simulations. Technical report, 2001.

[66] V. Dolean, P. Jolivet, and F. Nataf. *An Introduction to Domain Decomposition Methods: algorithms, theory and parallel implementation.* PhD thesis, France, 2015.

[67] Y. Dudai and K. Evers. To Simulate or Not to Simulate: What Are the Questions? *Neuron*, 84(2):254–261, 2014.

[68] C. Eliasmith, T. C. Stewart, X. Choo, T. Bekolay, T. Dewolf, Y. Tang, and D. Rasmussen. A Large-Scale Model of the Functioning Brain. *Science*, 338(i):1202–1205, 2012.

[69] J. M. Eppler. *A Multithreaded and Distributed System for the Simulation of Large Biological Neural Networks.* PhD thesis, Albert-Ludwigs-University Freiburg, 2006.

[70] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of the Internet Topology. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, volume 50, pages 251–262, Cambridge (USA), 1999.

[71] C. Fernandez-Musoles. Towards Neuroimaging real-time driving using Convolutional Neural Networks. In *Computer Science and Electronic Engineering (CEEC), 2016 8th*, 2016.

[72] C. Fernandez-Musoles, D. Coca, and P. Richmond. Communication optimisation in distributed Spiking Neural Network simulations. In *2ND HBP Student Conference - Transdisciplinary Research Linking Neuroscience, Brain Medicine, and Computer Science*, pages 21–23, Ljubljana (Slovenia), 2017.

[73] C. Fernandez-Musoles, D. Coca, and P. Richmond. Communication Sparsity in Distributed Spiking Neural Network Simulations to Improve Scalability. *Frontiers in Neuroinformatics*, (April):1–15, 2019.

[74] C. Fernandez-Musoles, D. Coca, and P. Richmond. HyperPRAW : Architecture-Aware Hypergraph Restreaming Partition to Improve Performance of Parallel Applications Running on High Performance Computing Systems. In *Proceedings of the 48th International Conference on Parallel Processing - ICPP 2019*, Kyoto (Japan), 2019.

[75] A. K. Fidjeland, E. B. Roesch, M. P. Shanahan, and W. Luk. NeMo: A platform for neural modelling of spiking neurons using GPUs. *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, pages 137–144, 2009.

[76] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Design Automation Conference*, pages 175–181, Las Vegas, Nevada (USA), 1982.

[77] J. Fieres, J. Schemmel, and K. Meier. in a Configurable Wafer-Scale Hardware System. *Neural Networks*, pages 969–976, 2008.

[78] H. Firth and P. Missier. Workload-aware streaming graph partitioning. In *Workshop Proceedings of the EDBT/ICDT 2016 Joint Conference (March*, Bordeaux, France, 2016.

[79] P. J. Fox. *Massively Parallel Neural Computation*. PhD thesis, University of Cambridge, 2013.

[80] R. Fujimoto. Research Challenges in Parallel and Distributed Simulation. *ACM Transactions on Modeling and Computer Simulation*, 26(4):1–29, 2016.

[81] Philippe Galinier, Zied Boujbel, and Michael Coutinho Fernandes. An efficient memetic algorithm for the graph partitioning problem. *Annals of Operations Research*, 191(1):1–22, 2011.

[82] F. Galluppi, S. Davies, A. Rast, T. Sharp, L. A. Plana, and S. Furber. A hierachical configuration system for a massively parallel neural hardware platform. In *Proceedings of the 9th conference on Computing Frontiers - CF '12*, pages 183–192, Cagliari (Italy), 2012.

[83] M.R. Garey and D.S. Johnson. Computer and intractability. *A Series of Books in the Mathematical Sciences*, 1979.

[84] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, pages 1299–1308, 2013.

[85] W. Gerstner and W. M Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. 2002.

[86] M. Gewaltig and M. Diesmann. NEST (Neural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007.

[87] G. Gill, R. Dathathri, L. Hoang, and K. Pingali. A study of partitioning policies for graph analytics on largescale distributed platforms. *Proceedings of the VLDB Endowment*, 12(4):321–334, 2018.

[88] L. Givon. *An Open Pipeline for Generating Executable Neural Circuits from Fruit Fly Brain Data*. PhD thesis, Columbia University, 2016.

[89] L. Givon and A. A. Lazar. Neurokernel: An open source platform for emulating the fruit fly brain. *PLoS ONE*, 11(1), 2015.

[90] J. E. Gonzalez, Y. Low, and H. Gu. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 17–30, Hollywood, USA, 2012.

[91] D. Goodman and R. Brette. Brian: a simulator for spiking neural networks in python. *Frontiers in neuroinformatics*, 2(November):5, 2008.

[92] D. Goodman and R. Brette. The brian simulator. *Frontiers in Neuroscience*, 3(SEP):192–197, 2009.

[93] S. Grillner, N. Ip, C. Koch, W. Koroshetz, H. Okano, M. Polachek, M. Poo, and T. Stejnowski. Worldwide initiatives to advance brain research. *Nature Neuroscience*, 19(9):1118–1122, 2016.

[94] W. D. Gropp. Parallel Computing and Domain Decomposition. In *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, 1992.

[95] I. A. Guney, B. S. Ovant, and S. Baydere. Impact of RDMA Communication on the Performance of Distributed BFS Algorithm. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 350–356, 2016.

[96] J. Hahne, M. Helias, S. Kunkel, J. Igarashi, M. Bolten, A. Frommer, and M. Diesmann. A unified framework for spiking and gap-junction interactions in distributed neuronal network simulations. *Frontiers in neuroinformatics*, 9(September):22, 2015.

[97] P. Hammarlund and O. Ekeberg. Large neural network simulations on multiple hardware platforms. *Journal of Computational Neuroscience*, 5(4):443–459, 1998.

[98] P. Hammarlund and O. Ekeberg. Large neural network simulations on multiple hardware platforms. *Journal of Computational Neuroscience*, 5(4):443–459, 1998.

[99] J. Hasler and B. Marr. Finding a roadmap to achieve large neuromorphic hardware systems. *Frontiers in Neuroscience*, 7(7 SEP):1–29, 2013.

[100] T. Hatazaki. Rank Reordering Strategy for MPI Topology Creation Functions. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 188–195, 1998.

[101] E. M. Heien, Y. Asai, and T. Nomura. Optimization Techniques for Parallel Biophysical Simulations Generated by insilicoIDE. *Computing Systems*, 2(2):131–143, 2009.

[102] E. M. Heien, M. Okita, Y. Asai, T. Nomura, and K. Hagihara. insilicoSim : an Extendable Engine for Parallel Heterogeneous Biophysical Simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 1–10, Torremolinos, Malaga (Spain), 2010.

[103] B. A. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '95*, pages 28–es, San Diego, California (USA), 1995.

[104] M. L. Hines, H. Eichner, and F. Schürmann. Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. *Journal of Computational Neuroscience*, 25(1):203–210, 2008.

[105] M. L. Hines, H. Markram, and F. Schürmann. Fully implicit parallel simulation of single neurons. *Journal of Computational Neuroscience*, 25(3):439–448, 2008.

[106] R. V. Hoang, D. Tanna, L. C. Jayet Bray, S. M. Dascalu, and F. C Harris. A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling. *Frontiers in neuroinformatics*, 7(October):19, 2013.

[107] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.

[108] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. De Supinski, R. Thakur, and J. L. Träff. The scalable process topology interface of MPI 2.2. *Concurrency Computation Practice and Experience*, 23:293–310, 2010.

[109] T. Hoefler and T. Schneider. Optimization principles for collective neighborhood communications. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, Salt Lake City, USA, 2012.

[110] T. Hoefler, T. Schneider, and A. Lumsdaine. Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale. *International Journal of Parallel, Emergent and Distributed Systems*, 25(4):241–258, 2010.

[111] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 45, page 159, Bangalore (India), 2010.

[112] T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. *Proceedings of the international conference on Supercomputing - ICS '11*, page 75, 2011.

[113] T. Hoefler and J. L. Träff. Sparse collective operations for MPI. In *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, Rome (Italy), 2009.

[114] T. Hu, H. Xiong, W. Zhou, S. Y. Sung, and H. Luo. Hypergraph partitioning for document clustering: A unified clique perspective. In *ACM SIGIR 2008 - 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Proceedings*, pages 871–872, Singapore, 2008.

[115] S. Huang, E. Aubanel, and V. C. Bhavsar. PaGrid: A mesh partitioner for computational grids. *Journal of Grid Computing*, 4(1):71–88, 2006.

[116] D. Ibanez. A Simple C Api for Dynamic Sparse Exchange. Technical report, Scientific Computation Research Center, Rensselaer Polytechnic Institute, New York, 2015.

[117] E. Ihler, D. Wagner, and F. Wagner. Modeling hypergraphs by graphs with the same mincut properties. *Information Processing Letters*, 45:171–175, 1993.

[118] G. Indiveri, B. Linares-Barranco, T. J. Hamilton, A. van Schaik, R. Etienne-Cummings, T. Delbruck, S. C. Liu, P. Dudek, P. HAfliger, S. Renaud, J. Schemmel, G. Cauwenberghs, J. Arthur, K. Hynna, F. Folowosele, S. Saighi, T. Serrano-Gotarredona, J. Wijekoon, Y. Wang, and K. Boahen. Neuromorphic silicon neuron circuits. *Frontiers in Neuroscience*, 5(MAY):1–23, 2011.

[119] E. M. Izhikevich and G. M. Edelman. Large-scale model of mammalian thalamocortical systems. *Proceedings of the National Academy of Sciences*, 105(9):3593–3598, 2008.

[120] F. Jacobacci, M. Sapir, S. Collavini, S. Kochen, and A. Blenkmann. Assessing effective connectivity in epileptogenic networks. A model-based simulation approach. *Journal of Physics: Conference Series*, 477:012037, 2013.

[121] E. Jeannot and G. Mercier. Near-optimal placement of MPI processes on hierarchical NUMA architectures. *Euro-Par 2010 - Parallel Processing*, 6272:199–210, 2010.

[122] W. Jiang, J. Qi, J. Xu Yu, J. Huang, and R. Zhang. HyperX: A Scalable Hypergraph Framework. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):909–922, 2018.

[123] X. Jin, F. Galluppi, C. Patterson, A. Rast, S. Davies, S. Temple, and S. Furber. Algorithm and software for simulation of spiking neural networks on the multi-chip SpiNNaker system. *Proceedings of the International Joint Conference on Neural Networks*, 2010.

[124] J. Jordan, T. Ippen, M. Helias, I. Kitayama, M. Sato, J. Igarashi, M. Diesmann, and S. Kunkel. Extremely Scalable Spiking Neuronal Network Simulation Code: From Laptops to Exascale Computers. *Frontiers in Neuroinformatics*, 12(February), 2018.

[125] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, and A. Shalita. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. *Proceedings of the VLDB Endowment 2017*, 10(11):1418–1429, 2017.

[126] E. R. Kandel, H. Markram, P.M. Matthews, R. Yuste, and V. Koch. Neuroscience thinks big (and collaboratively). *Nature reviews*, 14(9):659–64, 2013.

[127] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[128] G. Karypis and V. Kumar. Multilevel k -way Hypergraph Partitioning. In *36th Design Automation Conference*, volume 11, pages 285 – 300, New Orleans (USA), 1999.

[129] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49:291–307, 1970.

[130] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber. SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor. *Proceedings of the International Joint Conference on Neural Networks*, pages 2849–2856, 2008.

[131] T. Kiefer, D. Habich, and W. Lehner. Penalized Graph Partitioning for Static and Dynamic Load Balancing. In *Euro-Par 2016 : Parallel processing*, pages 146–158, 2016.

[132] J. Knight and T. Nowotny. Larger GPU-accelerated brain simulations with procedural connectivity. *bioRxiv*, 2020.

[133] C. Koch. Project MindScope. In *Frontiers in Computational Neuroscience*, Munich (Germany), 2012.

[134] J. Kozloski and J. Wagner. An Ultrascalable Solution to Large-scale Neural Tissue Simulation. *Frontiers in Neuroinformatics*, 5(September), 2011.

[135] S. Kumar, P. Heidelberg, D. Chen, and M. L. Hines. Optimization of Applications with Non-blocking Neighborhood Collectives via Multisends on the Blue Gene/P Supercomputer. *2010 IEEE International Symposium on Parallel & Distributed Processing*, 23:1–11, 2010.

[136] S. Kunkel, T. C. Potjans, J. M. Eppler, H. E. Plesser, A. Morrison, and M. Diesmann. Meeting the memory challenges of brain-scale network simulation. *Frontiers in Neuroinformatics*, 5(January):35, 2012.

[137] S. Kunkel, M. Schmidt, J. M. Eppler, H. E. Plesser, G. Masumoto, J. Igarashi, S. Ishii, T. Fukai, A. Morrison, M. Diesmann, and M. Helias. Spiking network simulation code for petascale computers. *Frontiers in neuroinformatics*, 8(October):78, 2014.

[138] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, pages 225–236, 2013.

[139] A. A. Lazar, K. Psychas, N. H. Ukani, and Y. Zhou. A Parallel Processing Model of the Drosophila Retina. *Neurokernel RFC*, 3(1):1–52, 2015.

[140] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, 1990.

[141] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. PyTorch-BigGraph: A Large-scale Graph Embedding System. *arXiv:1903.12287v3 [cs.LG]*, 2019.

[142] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *arXiv:0810.1355 [cs.DS]*, 6(1):29–123, 2009.

[143] H. Li, Z. Chen, R. Gupta, and M. Xie. Non-intrusively avoiding scaling problems in and out of MPI collectives. *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018*, pages 415–424, 2018.

[144] Q. Liu, Y. Huang, and D. N. Metaxas. Hypergraph with sampling for image retrieval. *Pattern Recognition*, 44(10-11):2255–2262, 2011.

[145] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *Proceedings ofthe VLDB Endowment*, volume 5, pages 716–727, Istambul, Turkey, 2012.

[146] W. W. Lytton. Computer modelling of epilepsy. *Nature reviews*, 9(8):626–637, 2008.

[147] W. W. Lytton, A. Seidenstein, S. Dura-Bernal, R. McDougal, F. Schurmann, and M. L. Hines. Simulation Neurotechnologies for Advancing Brain Research: Parallelizing Large Networks in NEURON. *Neural Computation*, 28(10):2063–2090, 2016.

[148] L. P. Maguire, T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin. Challenges for large-scale implementations of spiking neural networks on FPGAs. *Neurocomputing*, 71(1-3):13–29, 2007.

[149] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *Proceedings CCGRID 2008 - 8th IEEE International Symposium on Cluster Computing and the Grid*, pages 130–137, Lyon (France), 2008.

[150] H. Markram. The Human Brain Project. Technical report, European Commission, 2012.

[151] B. Marr, B. Degnan, P. Hasler, and D. Anderson. Scaling energy per operation via an asynchronous pipeline. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(1):147–151, 2013.

[152] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, and K. Rothermel. HYPE: Massive Hypergraph Partitioning with Neighborhood Expansion. In *Proceedings of 2018 IEEE International Conference on Big Data (Big Data)*, pages 458–467, Seattle, USA, 2018.

[153] C. Mayer, R. Mayer, M. A. Tariq, H. Geppert, L. Laich, L. Rieger, and K. Rothermel. ADWISE: Adaptive Window-based Streaming Edge Partitioning for High-Speed Graph Processing. *arXiv:1712.08367v1 [cs.DC]*, 2017.

[154] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel. GrapH: Heterogeneity-Aware Graph Computation with Adaptive Partitioning. In *2016 IEEE 36th International Conference on Distributed Computing Systems*, pages 118–128, 2016.

[155] J. McCalpin. Memory Bandwidth and System Balance in HPC Systems. In *Super Computing 2016 Talk*, Salt Lake City, 2016.

[156] A. McGregor. Graph Stream Algorithms: A Survey. *ACM Sigmod record*, 43(1):9–20, 2014.

[157] G. Mercier and J. Clet-Ortega. Towards an efficient process placement policy for MPI applications in multicore environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759, pages 104–115, 2009.

[158] M. Migliore, C. Cannia, W. W. Lytton, H. Markram, and M. L. Hines. Parallel Network Simulations with NEURON. *Journal of Computer Neuroscience*, 21(2):119–129, 2006.

[159] K. Minkovich, C. M. Thibeault, M. J. O'Brien, A. Nogin, Y. Cho, and N. Srinivasa. HRLSim: A high performance spiking neural network simulator for GPGPU clusters. *IEEE Transactions on Neural Networks and Learning Systems*, 25(2):316–331, 2014.

[160] A. Morrison, C. Mehring, T. Geisel, A.D. Aertsen, and M. Diesmann. Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural computation*, 17(8):1776–1801, 2005.

[161] I. Moulitsas and G. Karypis. Architecture aware partitioning algorithms. *International Conference on Algorithms and Architectures for Parallel Processing*, 5022:42–53, 2008.

[162] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H J Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. *Innovative Parallel Computing, InPar 2012*, (2), 2012.

[163] J. Mutch, U. Knoblich, and T. Poggio. CNS: a GPU-based framework for simulating cortically-organized networks. *Computer Science and Artificial Intelligence Laboratory Technical Report*, (2010-02-26), 2010.

[164] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. Veidenbaum. Efficient simulation of large-scale Spiking Neural Networks using CUDA graphics processors. In *2009 International Joint Conference on Neural Networks*, pages 2145–2152, Atlanta, GA (USA), 2009.

[165] M. E. J. Newman. Algorithms for graph partitioning: A survey. *Linkoping Electronic Articles in Computer and Information Science*, 3(10), 1998.

[166] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. HERMES: Dynamic partitioning for distributed social network graph databases. In *EDBT 2015 - 18th International Conference on Extending Database Technology, Proceedings*, pages 25–36, Brussels, Belgium, 2015.

[167] C. Nugteren, G. Braak, and H. Corporaal. A Study of the Potential of Locality-Aware Thread Scheduling for GPUs. In *Euro-Par 2014: Parallel Processing Workshops*, pages 146–157, 2014.

[168] A. Pantazi, S. Woźniak, T. Tuma, and E. Eleftheriou. All-memristive neuromorphic computing with level-tuned neurons. *Nanotechnology*, 27(35):355205, 2016.

[169] M. Patwary, S. Garg, and B. Kang. Window-based Streaming Graph Partitioning Algorithm. *ACM International Conference Proceeding Series*, pages 1–16, 2019.

[170] G. A. Pavlopoulos, M. Secrier, C. N. Moschopoulos, T. G. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. G. Bagos. Using graph theory to analyze biological networks. *BioData Mining*, 4(1):1–27, 2011.

[171] D. Pecevski, T. Natschläger, and K. Schuch. PCSIM: A Parallel Simulation Environment for Neural Circuits Fully Integrated with Python. *Frontiers in Neuroinformatics*, 3(May):15, 2009.

[172] S. V. Pericherla and S. Vadhiyar. High Performance and Enhanced Scalability for Parallel Applications using MPI-3's non-blocking Collectives. *Procedia Computer Science*, 108:2403–2407, 2017.

[173] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. HDRF : Stream-Based Partitioning for Power-Law Graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 243–252, Melbourne, Australia, 2015.

[174] H. E. Plesser, J. M. Eppler, A. Morrison, M. Diesmann, and M. Gewaltig. Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers. *Neuroscience*, 4641:672–681, 2007.

[175] M. Poo, J. Du, N. Y. Ip, Z. Xiong, B. Xu, and T. Tan. China Brain Project: Basic Neuroscience, Brain Diseases, and Brain-Inspired Computing. *Neuron*, 92(3):591–596, 2016.

[176] A. Pothen, H. D. Simon, and K. P. Liu. Partitioning Sparse Matrices with Eigenvectors of Graphs. Technical report, NAS Systems Division, 1989.

[177] T. C. Potjans and M. Diesmann. The cell-type specific cortical microcircuit: Relating structure and activity in a full-scale spiking network model. *Cerebral Cortex*, 24(3):785–806, 2014.

[178] W. Potjans, A. Morrison, and M. Diesmann. Enabling functional neural circuit simulations with distributed computing of neuromodulated plasticity. *Frontiers in computational neuroscience*, 4(November):141, 2010.

[179] S. Pronk, S. Pall, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, Peter M. Kasson, D. Van Der Spoel, B. Hess, and E. Lindahl. GROMACS 4.5: A high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[180] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. A Distributed Algorithm for Large-Scale Graph Partitioning. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(2), 2015.

[181] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp. Multi-core and network aware MPI topology functions. In *EuroMPI 2011: Recent Advances in the Message Passing Interface*, volume 6960, pages 50–60, 2011.

[182] A. Rast, F. Galluppi, S. Davies, L. Plana, C. Patterson, T. Sharp, D. Lester, and S. Furber. Concurrent heterogeneous neural model simulation on real-time neuromimetic hardware. *Neural Networks*, 24(9):961–978, 2011.

[183] A. Rey, F. D. Igual, and M. Prieto-Matias. HeSP: A simulation framework for solving the task scheduling-partitioning problem on heterogeneous architectures. In *Euro-Par 2016: Parallel Processing*, volume 9833, pages 183–195, 2016.

[184] E. R. Rodrigues, F. L. Madruga, P. O A Navaux, and J. Panetta. Multi-Core aware process mapping and its impact on communication overhead of parallel applications. *Proceedings - IEEE Symposium on Computers and Communications*, pages 811–817, 2009.

[185] A. Rosado-Muñoz, M. Bataller-Mompean, and J. Guerrero-Martinez. FPGA implementation of Spiking Neural Networks. In *Proceedings of the 1st IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control - CESCIT*, pages 139–144. IFAC, 2012.

[186] C. Rossant, D. Goodman, B. Fontaine, J. Platkiewicz, A. K. Magnusson, and R. Brette. Fitting neuron models to spike trains. *Frontiers in Neuroscience*, 5(FEB):1–8, 2011.

[187] A. Rousset, B. Herrmann, C. Lang, and L. Philippe. A survey on parallel and distributed Multi-Agent Systems. *Computer Science Review*, 22(November):27–46, 2016.

[188] X. Rubio-Campillo. Pandora : A Versatile Agent-Based Modelling Platform for Social Simulation. *Proceedings of the 6th International Conference on Advances in System Simulation*, (c):29–34, 2014.

[189] H. P. Sajjad, A. H. Payberah, F. Rahimian, V. Vlassov, and S. Haridi. Boosting vertex-cut partitioning for streaming graphs. In *Proceedings - 2016 IEEE International Congress on Big Data*, San Francisco, USA, 2016.

[190] S. N. Satchidanand, H. Ananthapadmanaban, and B. Ravindran. Extended discriminative random walk: A hypergraph approach to multi-view multi-relational transductive learning. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015) Extended*, pages 3791–3797, Buenos Aires (ARG), 2015.

[191] A. Sboev, D. Vlasov, A. Serenko, R. Rybka, and I. Moloshnikov. A comparison of learning abilities of spiking networks with different spike timing-dependent plasticity forms. *Journal of Physics: Conference Series*, 681(1):012013, 2016.

[192] W. Schenck, Y. V. Zaytsev, A. Morrison, A. V. Adinetz, and D. Pleiter. Performance Model for Large – Scale Neural Simulations with NEST. In *SC14 Conference for Supercomputing (Extended Poster Abstracts)*, New Orleans (USA), 2014.

[193] S. Schlag. A Benchmark Set for Multilevel Hypergraph Partitioning Algorithms [Data set], 2017.

[194] S. Schlag, C. Schulz, D. Seemaier, and D. Strash. Scalable edge partitioning. *Proceedings of the Workshop on Algorithm Engineering and Experiments*, January:211–225, 2019.

[195] M. Schmidt, R. Bakker, C. C. Hilgetag, M. Diesmann, and S. J. van Albada. Multi-scale account of the network structure of macaque visual cortex. *Brain Structure and Function*, 223(3):1409–1435, 2018.

[196] S. Schmitt, J. Klaehn, G. Bellec, A. Gruebl, M. Guettler, A. Hartel, S. Hartmann, D. Husmann, K. Husmann, V. Karasenko, M. Kleider, C. Koke, C. Mauch, E. Mueller, P. Mueller, J. Partzsch, M. A. Petrovici, S. Schiefer, S. Scholze, B. Vogginger, R. Legenstein, W. Maass, C. Mayr, J. Schemmel, and K. Meier. Neuromorphic Hardware in the Loop: Training a deep spiking network on the BrainScaleS wafer-scale system. *International Joint Conference on Neural Networks (IJCNN)*, pages 2227–2234, 2017.

[197] O. Selvitopi, S. Acer, and C. Aykanat. A Recursive Hypergraph Bipartitioning Framework for Reducing Bandwidth and Latency Costs Simultaneously. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):345–358, 2017.

[198] R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodríguez, L. Camunas-Mesa, R. Berner, M. Rivas-Perez, T. Delbrück, S. C. Liu, R. Douglas, P. Hafliger, G. Jimenez-Moreno, A. Civit Ballcels, T. Serrano-Gotarredona, A. J. Acosta-Jimenez, and B. Linares-Barranco. CAVIAR: A 45k neuron, 5M synapse, 12G connects/s AER hardware sensory-processing-learning-actuating system for high-speed visual object recognition and tracking. *IEEE Transactions on Neural Networks*, 20(9):1417–1438, 2009.

[199] T. Sharp, F. Galluppi, A. Rast, and S. Furber. Power-efficient simulation of detailed cortical microcircuits on SpiNNaker. *Journal of Neuroscience Methods*, 210(1):110–118, 2012.

[200] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2-3):135–148, 1991.

[201] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra. Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration. *Proceedings of the International Conference on Parallel Processing*, pages 228–237, 2010.

[202] T. C. Stewart. A Technical Overview of the Neural Engineering Framework. *AISB Quarterly: The Newsletter of the Society for the Study of Artificial Intelligence and Simulation of Behaviour*, (135), 2012.

[203] T. C. Stewart, B. Tripp, C. Eliasmith, S. TC, T. B, and E. C. Python scripting in the nengo simulator. *Frontiers in neuroinformatics*, 3(March):7, 2009.

[204] P. Szynkiewicz. A novel GPU-enabled simulator for large scale spiking neural networks. *Journal of Telecommunications and Information Technology*, 2016(2):34–42, 2016.

[205] J. D. Teresco, J. Faik, and J. E. Flaherty. Hierarchical partitioning and dynamic load balancing for scientific computation. *Applied Parallel Computing. State of the Art in Scientific Computing*, 3732 LNCS:911–920, 2004.

[206] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Träff. MPI at Exascale. In *Proceedings of SciDAC*, pages 14–35, Chattanooga, Tennessee (USA), 2010.

[207] S. Theil. Why the Human Brain Project Went Wrong—and How to Fix It. *Scientific Americal*, pages 1–15, 2015.

[208] C. M. Thibeault, R. V. Hoang, and F. C. Harris. A Novel Multi-GPU Neural Simulator. In *Conference: Proceedings of the ISCA 3rd International Conference on Bioinformatics and Computational Biology*, number 1, 2011.

[209] J. L. Träff. Implementing the MPI Process Topology Mechanism. *Supercomputing, ACM/IEEE 2002 Conference*, page 28, 2002.

[210] A. Trifunovic. *Parallel Algorithms for Hypergraph Partitioning*. PhD thesis, University of London, 2006.

[211] A. Trifunovic and W. Knottenbelt. Parallel Multilevel Algorithms for Multiconstraint Graph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563 – 581, 2008.

[212] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL : Streaming Graph Partitioning for Massive Scale Graphs Categories and Subject Descriptors. Technical report, 2012.

[213] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.

[214] G. Urgese, F. Barchi, E. Macii, and A. Acquaviva. Optimizing Network Traffic for Spiking Neural Network Simulations on Densely Interconnected Many-Core Neuromorphic Platforms. *IEEE Transactions on Emerging Topics in Computing*, 6750(c):1–1, 2016.

[215] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J J Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. De Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[216] Sacha J. van Albada, Andrew G. Rowley, Johanna Senk, Michael Hopkins, Maximilian Schmidt, Alan B. Stokes, David R. Lester, Markus Diesmann, and Steve B. Furber. Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model. *Frontiers in Neuroscience*, 12(MAY):1–20, 2018.

[217] J. Vitay, H. Ü. Dinkelbach, and F. H. Hamker. ANNarchy: a code generation approach to neural simulations on parallel hardware. *Frontiers in neuroinformatics*, 9(July):19, 2015.

[218] T. P. Vogels and L. F. Abbott. Signal Propagation and Logic Gating in Networks of Integrate-and-Fire Neurons. *The Journal of Neuroscience*, 25(46):10786–10795, 2005.

[219] C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Computer Systems*, 17(5):601–623, 2001.

[220] X. Wang, L. Tang, H. Gao, and H. Liu. Discovering overlapping groups in social media. In *The 10th IEEE International Conference on Data Mining*, pages 569–578, Sidney (Australia), 2010.

[221] Y. Wang, P. Li, and C. Yao. Hypergraph canonical correlation analysis for multi-label classification. *Signal Processing*, 105:258–267, 2014.

[222] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. *arXiv:1402.2394v1 [cs.DB]*, 2014.

[223] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao. Heterogeneous Environment Aware Streaming Graph Partitioning. *IEEE Transactions on Knowledge and Data Engineering*, 27(6):1560–1572, 2015.

[224] Z. Xu and K. Hwang. Modeling communication overhead: MPI and MPL performance on the IBM SP2. *IEEE Parallel and Distributed Technology*, 4(1):9–23, 1996.

[225] J. Xue, Z. Yang, S. Hou, and Y. Dai. When computing meets heterogeneous cluster: Workload assignment in graph computation. *Proceedings - 2015 IEEE International Conference on Big Data*, pages 154–163, 2015.

[226] W. Yang, G. Wang, K. K. R. Choo, and S. Chen. HEPart : A balanced hypergraph partitioning algorithm for big data applications. *Future Generation Computer Systems*, 83:250–268, 2018.

[227] W. Yang, G. Wang, L. Ma, and S. Wu. A Distributed Algorithm for Balanced Hypergraph Partitioning. *Advances in Services Computing. APSCC 2016. Lecture Notes in Computer Science*, 10065:477–490, 2016.

[228] E. Yavuz, J. Turner, and T. Nowotny. GeNN: a code generation framework for accelerated brain simulations. *Nature: Scientific reports*, 6(January), 2016.

[229] F. Zenke and W. Gerstner. Limits to high-speed simulations of spiking neural networks using general-purpose computers. *Frontiers in neuroinformatics*, 8(September):76, 2014.

[230] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li. Graph edge partitioning via neighborhood heuristic. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, volume Part F1296, pages 605–614, Halifax, Canada, 2017.

[231] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Architecture-aware graph repartitioning for data-intensive scientific computing. *Proceedings - 2014 IEEE International Conference on Big Data*, pages 78–85, 2014.

[232] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Planar: Parallel lightweight architecture-aware adaptive graph repartitioning. *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016*, (June):121–132, 2016.

[233] A. Zheng, A. Labrinidis, P. K. Chrysanthis, and J. Lange. ARGO : Architecture-Aware Graph Partitioning. In *2016 IEEE International Conference on Big Dat*, pages 284–293, 2016.

[234] A. Zheng, A. Labrinidis, P. Pisciuneri, P. K. Chrysanthis, and P. Givi. PARAGON: Parallel Architecture-Aware Graph Partition Refinement Algorithm. In *19th International Conference on Extending Database Technology (EDBT*, pages 365–376, 2016.

Appendix A

Using neuronal activity to improve communication optimisation

Work published in Frontiers Event Abstracts for the 2nd Human Brain Project Student Conference [72]. It demonstrates the potential gains to be made by incorporating runtime simulation communication profiling to improve workload allocation by reducing communication overhead.

Introduction/Motivation: In distributed computation, higher parallelism is desira- ble to split the computation into independently executable parts. Ideally such parts do not require any synchronisation to speed-up execution time. In Spiking Neuron Network (SNN) simulations, this is not realistic due to the high level of interconnectivity amongst neurons. Thus, higher parallelism increases communication overhead, limiting scalability [229].

There are two ways of mitigating parallel overhead in neuronal simulations: commu- nicating more efficiently (targeted propagation); and decreasing the amount of data to be sent (number of spikes going across partitions). This work proposes solutions to the scale of communication at both levels.

Methods: Sending every spike to all processes has been shown to scale poorly for large parallel sims and be wasteful because not all partitions require all spike data [134, 135]. Thus, by only sending the relevant data to the interested processes, communication volume can be reduced. Two alternative point-to-point strategies are proposed, based on how the inter-process messages are coordinated at each time step.

Parallel simulators that have considered the mapping of neurons to processes have focused on computational load balance alone [5]. To date, state-of-the-art parallel sim- ulators are not considering communication amongst neurons to inform the mapping of neurons to processes. Authors have suggested the impact of neuron connectivity would have [159, 214], but not on actual network activity. The proposed approach models the SNN as a graph, where the vertices (neurons) are weighted proportional to their activity during simulation (neurons with high activity spike frequently, hence com- municating more often with post-synaptic neurons). Multi-level k-way partitioning is used to minimise the volume of communication between vertices and map them to processes. To gather results, simulations of a Cortical Microcircuit model [177] are performed across both experiments.

Results and Discussion: Figure A.1A shows poor scalability of the collective all-to-all communication strategy and the reduction in communication volume when

using point-to-point strategies with respect to an all to all pattern. Not only the communi- cation is reduced, but it scales linearly with the number of processes.

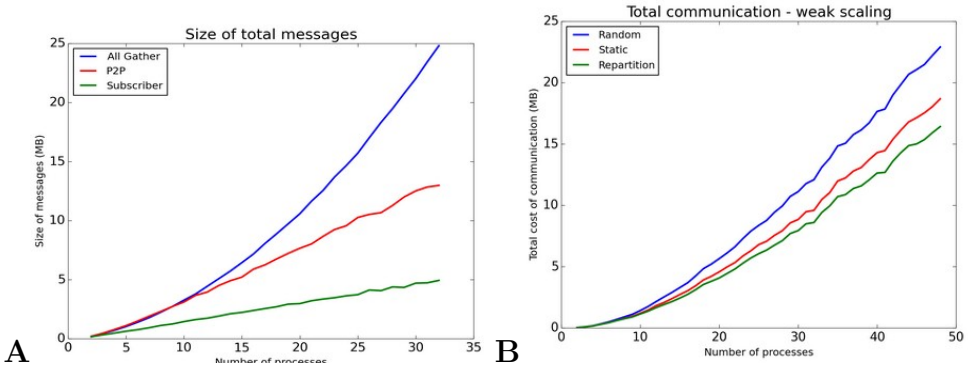


Figure A.1: **A**: Scaling of the number and weight of messages sent across during distributed simulations using three different communication strategies. **B**: Communication costs with different workload allocation strategies (SNN model is scaled with the number of processes)

The distribution of workload based on the communication volume of neurons is shown to reduce communication in distributed simulations in Figure A.1B. Two baselines are shown: random allocation and static partitioning (graph partitioning with equal weights edges and vertices). Repartitioning based on network activity shows an improvement of 40% over random and 12–15% over static partitioning.

As the total communication during simulation is reduced, simulations are expected to run faster, particularly given the scale of the communication volume in spiking neuron simulations. Further work could look into dynamic partitioning and graph analysis to inform partitioning in heterogeneous architectures (where processes can accept different workload).